Original

Software Testing and Verification

LEE J. WHITE

Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada

1. Introduction 335
1.1 Black Box Testing versus White Box Testing 337
1.2 Module, Integration, and Acceptance Testing 339
1.3 Instrumenting and Monitoring Programs for Tests 341
1.4 Categories of Errors 342
1.5 The Need for a Testing Oracle 343
2. Mathematical Theory of Testing 344
2.1 Graph Models 344
2.2 Fundamental Problems of Decidability in Testing 347
2.3 A Theoretical Foundation for Testing 350
3. Static Data Flow Analysis and Testing 352
3.1 Variable Anomalies 352
3.2 Data Flow Analysis 353
3.3 Critique of Static Analysis 354
4. Coverage Measures 354
4.1 Logic Coverage Measures 354
4.2 Iteration Loops 356
4.3 Data Flow Testing 356
5. Mutation Analysis and Functional Testing 358
5.1 What is Mutation Analysis? 359
5.2 Weak Mutation Testing—Another Approach 362
5.3 Functional Testing 364
6. Path-Oriented Testing Models 368
6.1 Path Testing Approaches 369
6.2 Domain Testing 371
6.3 The Sufficient Paths Criterion 377
6.4 Program Specification Testing and Partition Analysis 383
7. Conclusions and Future Prospects 387
7.1 Future Prospects 387

1. Introduction

Computer-program testing is currently a very active research field. There is the perception that formal program verification has a much more solid theoretical base, but is seldom utilized in practice. Testing is applied in virtually every software development project, yet the process is often not based upon a solid theoretical framework, even though it is expensive. This article will survey some of the basic concepts, approaches, and results in testing and will not necessarily be comprehensive or review all research methods. An excellent overview of software testing and validation techniques is provided in a tutorial format by Miller and Howden (1981). Texts on testing are available by Myers (1979) and by Beizer (1983).

Aside from the exact approach used for testing, there are two critical issues for the successful completion of a testing plan. A good software engineering methodology provides the required foundation for testing; this includes agreed-upon requirements, complete specifications, a systematic design methodology, walkthroughs and reviews, a structural code implementation, and a logical maintenance plan, which preserves the software structure, documentation, and test plan. The second critical issue is the necessity of articulating and agreeing upon the test plan early in the software life cycle. There should be agreement upon the test plan at the requirements stage, rather than frantically assembling this test plan as the software project approaches the implementation stage.

All of the software engineering methodology used provides a support and examination structure for the test plan, but no aspect is more important than the software specification. As this article will emphasize, the specification may be used to generate part of the test data, or may be used to decide whether or not the test data is correct. Thus the specification should be constructed with these or other testing functions in mind, since the software product is of questionable value if we cannot provide adequate confidence through testing.

In Section 2 we will consider the fact that no test strategy can be effective in detecting all errors in an arbitrary computer program. This does not preclude a test strategy being effective in detecting all errors for a specific program, but should provide an attitude of humility for both researcher and practitioner working with software verification. Any proposed testing methodology should be based upon scientifically sound principles, rather than an ad hoc approach, which is unfortunately the case far too often in contemporary practice. Thus it becomes important to involve the user with the test plan and testing process; if the objective of testing is to increase the confidence in the software quality, then it is imperative to describe the test plan to the user in simple terms. Too often the testing process only results in a large, incomprehensible file of test data.

Many software experts have argued that it is just as important to design software for easy maintenance and modification as it is to deliver that software with a minimum of errors. One of the reasons for this is the increased cost of maintenance and the extensive modifications many software systems will endure during this last but most important stage of the life cycle. The implications of this sort of extensive modifications of the software for testing are the following:

1. The specification of the software must also be modified. Any subsequent testing must be measured against this modified specification.
2. A modified test plan must also be constructed, to not only test the modified portions or modules of the software but also test interactions with other modules or portions of the software not directly affected by the changes.
3. The software must be entirely redesigned in order to incorporate the modifications. The reason for this is that the software characteristic of *structure* is designed by such modifications, and only through a redesign (not a "patch") can the integrity of structure be restored. Test plans and testing methodologies are in turn dependent upon this integrity of structure.
4. The additional testing should be carried out according to the modified test plan, added to a subset of the original test data on a test database with the same integrity and documentation as available when the software was first delivered.

Of course, it is precisely because precautions 1 through 4 are *not* observed in software maintenance that the process is so ineffective and costly. As is so true in other areas of endeavor, this level of investment at the beginning, whether it be in design or in maintenance, will pay dividends later through decreased cost at all stages of the software life cycle or during subsequent modifications.

This article will present a broad view of many important concepts needed for testing, either during design or under maintenance, together with an overview of a number of research results and approaches that will affect testing practice in the future. The object is to provide the practitioner with various models and methodologies that can be woven into a scientifically based test plan, rather than relying only upon an ad hoc approach.

1.1 Black Box Testing versus White Box Testing

One of the simplest ways to classify testing methods as "black box" or "white box" approaches. A *black box* testing approach will devise test data without any knowledge of the software under test or any aspect of its structure, whereas *white box* testing will explicitly use the program structure to develop test data.

1.1.1 Black Box Testing

Black box testing is often also known as *functional testing*, although in Section 5.3 we will describe an expanded version of functional testing, due to Howden (1985), which combines the black box and white box approaches.

One example of a strictly black box approach is to utilize the specifications to generate the test data. A concrete example of this approach is contained in Parnas (1972), where the objective of detailed specifications is to state the functional requirements completely and unambiguously, while leaving design decisions to the implementation. These specifications are then represented in the form of input–process–output, and can be conveniently used to generate test data. Another example of black box testing is *random test data generation*, where a random value is selected for each input variable of the program and each test data point then consists of these values collectively taken over all input variables. This is quite a simple and intuitively appealing approach, but it is unclear whether in general it will provide effective software error detection. Duran and Ntafos (1981) have reported experiments to show that random testing can be effective for at least some errors and classes of programs.

1.1.2 *White Box Testing*

A white box testing approach is based upon explicit knowledge of the software under test and its structure. For this reason, it is often also known as *structural testing*. Research has shown that there are inherent limitations to the use of white box testing alone, for test data based only upon the software code and structure will fail to detect the absence of certain features that might be missing in the software. Information and the associated test data must be derived from the software specifications, design documents, or from some other source. For example, a *missing path error* has been identified by Howden (1976), in which a required predicate does not appear in the given program to be tested. Especially if this missing predicate were an equality, it would be extremely difficult or impossible for a white box method to systematically determine that such a predicate should be present. We can only look to supplementary sources such as the software specification to provide information when such features might be missing within the given software.

Despite these disadvantages, a number of quite sophisticated and varied techniques have been developed and are presently used in both research and practice. One approach commonly used in practice involves *coverage measures*, including *statement coverage* and *branch coverage*. A detailed discussion of these techniques will be given in Section 4, but the basic idea is that a good test plan should certainly thoroughly exercise the various parts of the source code of the program. Thus, in statement coverage, each statement of the program should be executed for at least one input test point; in branch coverage each IF–THEN–ELSE predicate decision should have a true and a false outcome, and each of these should occur for at least one input test point. The simplicity of this approach is appealing, but it can be shown that many

errors will not be detected by these coverage methods and the software must be instrumented to establish the extent of coverage for the set of test data.

Path-oriented testing methods also constitute a white box approach. Here the process of testing a computer program is treated as two operations:

1. selection of a path or set of paths along which testing is to be conducted,
2. selection of input data to serve as test cases, which will cause the chosen paths to be executed.

Thus in the research literature there have been proposed methods to select paths for testing, or proposed methods to select data along program paths, without necessarily specifying an associated approach for selecting those paths; in other cases, proposed methods have addressed both aspects. Section 6 will discuss a number of path-oriented testing methods.

Substantial research has been done and viable methods proposed based upon structural testing. Fewer systematic studies, methods, and results are available in the area of black box testing. Howden (1985) has argued persuasively for a systematic functional testing approach, which utilizes information and generates test data from specifications, the software source code, and other design documents. This process of amalgamating the best of several methodologies is a reasonable approach.

1.2 Module, Integration, and Acceptance Testing

There are various levels at which software testing occurs. The lowest level takes place with the software *module*, which is the smallest unit of software. Many debates have raged as to the optimum or maximum size of the module, but most software experts agree that a software module should be of a size and complexity so as to be easily understood and grasped by a programmer or analyst other than its author. The levels of testing can then be identified as:

1. *module test* (also known as a *unit test*), where each module is individually tested to ensure that its performance meets its stated specification;
2. *integration test* (also known as an *incremental test*), where a set of modules are tested together, ensuring that the combined specifications of these modules are met as the modules interact and communicate; this requires careful testing of the interfaces between the modules in this set as they communicate and exchange information; these module interfaces are a common source of errors, especially if the modules are written by different programmers;
3. *systems test* (also known as an *evaluation test*), where the entire software

system is tested against the system specification—that is, all of the modules operating together; it is usually understood that this test is conducted by the software developer and reported to the user;

4. *acceptance test*, where a designated team performs a series of system tests on the delivered software and usually makes the acceptance of the software (and subsequent payment) contingent upon the successful completion of these tests; this designated team is independent from those who developed the product, and may be the user or a third party hired by the user.

In the following discussion, material is drawn extensively from Myers (1979), copyright © 1979, John Wiley & Sons, Inc., and is reprinted by permission of the publisher.

In the application of module tests, and to a greater extent for integration tests, additional software must be developed by the test analyst. For each module, where input data comes from other modules or external sources, a *test-driver module* must be developed to model that relationship and to provide test cases in an appropriate format. For modules that receive data from the module (or modules) under test, a *stub module* must be developed to model this relationship; this is particularly important and complex in the event that there is a transfer of control or that the stub must return appropriate values in response to the data initially communicated from the module (or modules) under test.

A difficult philosophical decision must be made by the test analyst in terms of the order in which module and integration testing should be conducted. One approach is *top-down testing*, where a main module is tested first. This is followed by integration tests involving modules called by or receiving data from this module, and this process continues until all modules are involved in a system test. For example, in Figure 1, if we identify module A as the primary module (the calling program), it should first be tested, but note that it will require stubs corresponding to modules B, C, and D. Subsequently B, C, and D can be tested using module A, but again stubs will be required for modules E, F, and G.

An alternative philosophy is *bottom-up testing*, in which the terminal modules in the system are tested first, which will require driver modules. Next, modules are tested that connect to the terminal modules, until the main module is included, culminating in the system test. In Figure 1, bottom-up testing would suggest first testing modules E, F, C, and G, requiring drivers for B (for E and F), A (for C), and D (for G). There are trade-offs between the top-down and bottom-up approaches, and there also exist hybrids of the two. For example, one needs to trade off the cost of drivers and stubs. Tests are generally easier to devise with bottom-up testing, but the main module is not involved until the end of the process, which could be a critical consideration. It is quite important to include these considerations in a test plan, for the costs of integration testing can become prohibitively high. A detailed discussion of these issues is given in Myers (1979).

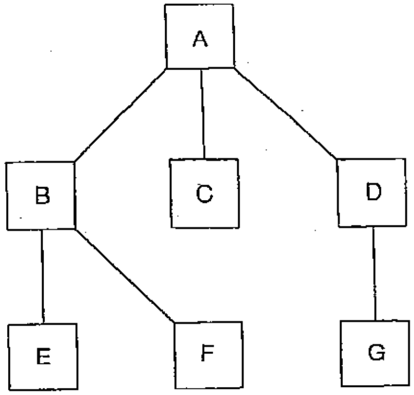

FIG. 1. Top-down versus bottom-up testing. (From Myers, reprinted by permission of John Wiley & Sons, Inc., copyright © 1979).

1.3 Instrumenting and Monitoring Programs for Tests

In program testing, input test data is typically submitted to the program and then the corresponding outputs are examined for errors. There exist, however, more explicit techniques for studying the nature and behavior of a program, by observing the progress of its execution. These are known as *probes* or *monitors*, and are generally software sensors inserted into the source code. There must be methods or aids for capturing, organizing, and analyzing the probe output. For example, if one uses statement coverage as an approach to generate appropriate test data, the software must be instrumented with statement counters to determine the extent of statement coverage.

There are a number of different software probes that can be used:

1. documentation probes
   - counters, which count executions of branches, code blocks, or logical program breaks
   - traces of variable values

- traces of procedure invocation and timing
- sensors for evaluating min/max/first/last values for loop control variables or code blocks

2. probes inserted to check for "standard" errors
   - division by zero
   - out-of-bounds array references
   - initialized variable references
   - procedure argument or parameter mismatches

Leon Stucki (1977) developed an extensive instrumentation system for FORTRAN programs, called Program Evaluator and Test (or PET), which utilized these concepts. Several researchers have investigated the optimum placement of probes and monitor software; for example, see the work of J. C. Huang (1975) or C. V. Ramamoorthy and K. H. Kim (1975).

Another powerful tool for testing is the incorporation of *assertions.* At various points in the program, the programmer who originally designed and wrote the source code is aware that certain conditions must hold at that point. As a matter of fact, one can argue that this is the sort of documentation, including intuitive loop invariants, that is really needed from the program author, rather than the documentation usually provided. The tester can utilize these conditions or assert that alternative conditions are useful for testing. An example might be

$$\text{ASSERT } (A + B) < 12$$

The violations of this assertion can be pointed out, but they also can be used to control various software instrumentation. An example of the use of such a dynamic assertion checker is again provided by the PET system of Stucki (1977).

1.4 Categories of Errors

In discussing the subject of testing, we should note that three terms are identified that are sometimes used interchangeably, but represent quite different phenomena. A *failure* in a system is an observable event where the system violates its specifications. An *error* is an item of information (such as a variable value or a line of code) which, when processed by the system, may produce a failure. Not every error will produce a failure, since errors may not be observable, or error recovery procedures may be built into the program. A *fault* is a mechanical or algorithmic defect that will generate an error (i.e., a programming "bug"). Note that our tendency is to refer to both errors and faults as errors.

Another concept related to program errors is that of *coincidental correctness.* This occurs when a fault is tested, and yet coincidentally the test data results in correct output variables. If different test data had exercised that fault, the fault would have been detected by the generation of incorrect output variables. This indicates that redundant testing is always required, and that tests should be designed with coincidental correctness in mind.

Howden (1976) originally defined the classification of errors into *domain* and *computation errors,* and this classification has subsequently proven useful. Zeil (1983) has published a more detailed classification and characterization of these concepts. A program is said to exhibit a *domain error* when incorrect output is generated due to execution of the wrong path through a program. A *computation error* occurs when the correct path through the program is taken, but the output is incorrect because of faults in the computations along that path.

There are two types of domain errors, *path selection errors* and *missing path errors.* When a path is incorrectly selected and another path exists that would produce correct output, we identify this as a *path selection error.* Where the conditional statement and computations associated with part of the input data domain are missing entirely, it is called a *missing path error* (Howden, 1976).

The simplest case of a path selection error is the *predicate fault*, in which a fault in a predicate causes execution to follow the wrong path for some input data. Since the evaluation of a predicate may depend upon previous assignment statements, a fault in an assignment statement can also cause domain errors. This will be referred to as an *assignment fault.*

1.5 The Need for a Testing Oracle

One of the most expensive aspects of testing is the simple determination of whether or not the output corresponding to test data is correct. What often happens in practice is that the tester "eyeballs" the output to see if it is reasonable, as correct output is known for only a small subset of the total test cases examined. Clearly a *test oracle* is needed that can automatically check the correctness of test output. A number of researchers have argued that the specifications should be executable so as to aid in this determination. The concept of a test oracle was originally developed by Howden (1978b), and he indicated that test oracles may also assume the form of tables of values, algorithms for hand computation, or formulas in the predicate calculus.

In practice, the programmer (or user) must make this determination, and the time spent examining and analyzing these test cases is a major factor in the high cost of software development. From a theoretical or research standpoint, we must assume the existence of a test oracle in order to refer to test cases as correct or incorrect.

2. Mathematical Theory of Testing

In order to study concepts of program structure, digraphs are introduced as a potential model. A control flow graph is defined and utilized for a number of concepts needed in program testing, together with the technique of data flow analysis. The analysis of the structure and data flow in computer programs is probably one of the best-understood areas of software engineering; many of these basic concepts were originally given in Hecht (1977).

One of the reasons that a comprehensive theory of testing has been so difficult is that so many of the fundamental questions in this area are found to be undecidable, i.e., unsolvable. A number of these results are reviewed, including the observation that the problem of selecting a reliable test-data set is unsolvable, and that no general testing strategy can be devised that will be effective for all programs.

The papers of Goodenough and Gerhart (1975) and of Howden (1976) are then discussed, as it is the opinion of most researchers in the area of program testing that these two papers comprise the fundamental basis for a theory of testing. Many of the concepts and definitions from those two papers are now used throughout the literature in a fundamental way, so these papers will be surveyed in this review. Many of the notions in this section were previously developed in White (1981), copyright © 1981 North-Holland Publishing Co.

2.1 Graph Models

Computer-program structure can be captured at the appropriate abstract level through the use of a *directed graph* (*digraph*). A digraph consists of a set of *nodes* and *arcs*, where an arc is a directed line between two nodes. To apply this digraph model to computer programs, the digraph should contain exactly one *entry node*, which has no incoming arcs, and should also contain exactly one *terminal node*, which has no arcs leaving it. Moreover, for every node in the digraph, there should exist a sequence of arcs such that this sequence can be traversed in the direction of the arcs from the entry node to that specified node. This sequence of arcs is called a *directed path*. Similarly, for every node in the digraph, there should also exist a directed path from that node to the terminal node of the digraph. Any digraph with these properties has been called a *well-formed digraph* by some authors (see, for example, Paige, 1975). Figure 2 shows an example of a control flow graph, together with the corresponding flow chart; note that most of the computational detail from the flow chart has been removed so that the emphasis is on the control flow. This control flow graph then appears as a digraph; we can determine whether or not it is well-formed.

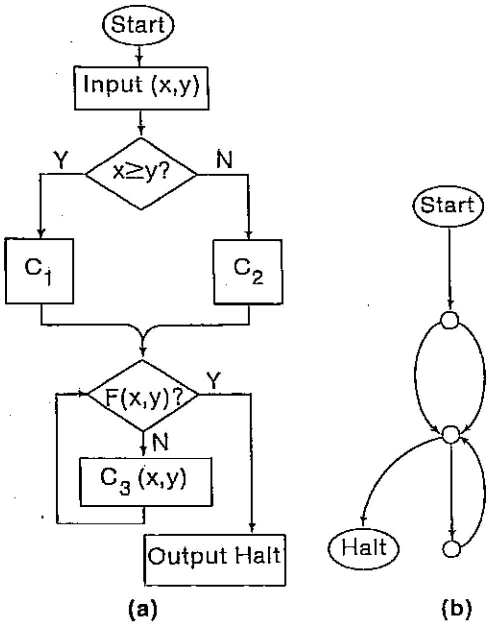

FIG. 2. Flow chart (a) and corresponding control flow graph (b).

2.1.1 *Control Flow Graphs*

In applying graph theory models of this type, the emphasis is placed on the decision points of a program, which determine the control flow, and there is less concern with the computational details and all the assignment statements of the program. For example, conditional transfers (such as the IF–THEN–ELSE construct) and the entry point of iteration loops (such as DO–WHILE or FOR loop constructs) constitute such decision points of a program where the flow of control may change depending upon the evaluation of a predicate expression. Although not recommended for use in well-designed programs ("structured programs"), another example of such a decision point is the occurrence of a GOTO construct.

We can now define a number of important concepts for program testing using this control flow graph model. A *control path* is defined to be a directed path from the entry node to the terminal node of the control flow graph. It should be noted that two directed paths that differ only in the number of times a particular iteration loop in the program is executed will be identified as distinct control paths. Thus the number of control paths in a program can be potentially infinite.

Every branch point of the program is associated with a general *predicate*. This predicate evaluates to true or false, and its value determines which outcome of the branch will be followed. A predicate is generated each time control reaches an IF or DO (or GOTO) statement in the given program. The *path condition* is the compound condition that must be satisfied by the input data point in order for the control path to be executed. The path condition consists of a set of constraints, one constraint for each predicate as it is encountered as the control path is traversed. The predicates are initially expressed in terms of program variables; since each of these program variables can be ultimately expressed in terms of the input variables using assignment statements along the control path, the predicates can be re-expressed as constraints in terms of only the input variables.

Not all the control paths that exist syntactically within the program are executable. If input data exist which satisfy the path condition, the control path is also an *execution path* and can be used in testing the program. If the path condition is not satisfied by any input value, the control path is said to be *infeasible* and is of no use in testing the program.

2.1.2 Static versus Dynamic Analysis

This brings to mind an important conceptual point when considering many issues in testing. *Static analysis* utilizes the computer program, examining this program for syntax errors and structural properties, but does not require execution of the program. *Dynamic analysis* requires execution of the program using input data. For example, one might be given a computer program in which the potential control flow is not well formed because no directed path from the entry node exists to reach some node of the program. This program defect can be detected by static analysis. Not every control path corresponds to an execution path, however, as some may prove to be infeasible, and this can only be detected by dynamic analysis. Another example of this distinction involves the concept of "reachable code." If there exists no control path by which to reach some specified set of code in a given program from the entry node (and thus the corresponding control flow graph is not well formed), this can be ascertained by static analysis. We say, however, that a statement (or a group of sequential statements) is *unreachable* if there exists no execution path that traverses that statement; again, this requires dynamic analysis to ascertain that this condition does or does not exist.

2.1.3 Symbolic Evaluation

Another approach used in testing is that of *symbolic evaluation*, which might also be thought of as another approach of static analysis, since the program is not actually executed with numerical data; an excellent survey of this area is provided by Clarke and Richardson (1981). The basic idea in symbolic evaluation is to allow the input variables to assume symbolic values and to express the output variables in terms of these symbols. These output variable expressions can then be examined to see if the program is computing the functions intended. One problem with this approach is that the computational functions, and thus the resultant symbolic output expressions, depend upon the path taken through the program. Thus, to effectively use symbolic evaluation, a specific control path must be selected, and the associated path condition obtained as a set of constraints to be satisfied must be expressed also in terms of the symbolic input variables. Then, for each control path, the computation can be explored by analyzing the symbolic output expression, and the feasibility of the path ascertained by determining that there exists at least one input point that satisfies the path condition. Many times the symbolic evaluation approach will show errors that might be difficult to determine using other methods. For this approach to be effective, the symbolic expressions should not be too complex; if they are too complex, their usefulness becomes limited.

2.2 Fundamental Problems of Decidability in Testing

One of the most important contributions of mathematics and computer science is the theory of decidability of computation. It has applications in many areas of computer science and is important in warning us not to attempt solutions to problems that can be proved unsolvable. In this section we will explore some of the decidability issues as applied to program testing, and it will become clear that many fundamental issues in testing are undecidable. This points out one more reason why the area of program testing is so difficult, and why approaches to testing have relied so heavily upon intuition, using heuristic and ad hoc techniques. Both the formal and informal aspects of the issues identified in this section are discussed by Brainerd and Landweber (1974).

A problem is said to be *undecidable* (or *unsolvable*) if it can be proved that no algorithm exists for its solution. One of the principal models for studying decidability issues is that of the *Turing machine*, which consists of a finite-state machine, together with an infinite-capacity external storage tape. A seminal result is that of the halting problem for the Turing machine, which is proved to be undecidable. The *halting problem* asks whether any given Turing machine will halt, given an arbitrary input.

If we are given another problem and wish to prove it undecidable, this may be accomplished by demonstrating that the decidability of the given problem implies the decidability of the halting problem (a contradiction). Almost all decidability proofs either directly or indirectly involve this method of reducing the halting problem to some other problem. It is almost immediate that it is undecidable whether a given computer program will halt, given an arbitrary input. In practical systems work, this problem is circumvented by carefully examining iteration loops to make sure they terminate under all conditions. The undecidability of this program halting problem means, however, that no computer system could be devised that will ascertain whether an arbitrary computer program will always terminate, given any input.

2.2.1 Computation of Equivalent Functions

A result with important implications for testing is that given two computer programs, the question as to whether they both compute the same function is undecidable (Brainerd and Landweber, 1974). If we identify one program as the "correct program" and consider the other program as the one we have before us which must be tested, this illustrates that even the availability of the "correct program" will not allow us in general to devise entirely reliable test data. For if we could prove the given program correct algorithmically by test data selection, the two programs would then be equivalent (compute the same function).

Another important application of this decidability result for testing involves control paths. If we ask in general whether two control paths from the same program, or from different programs, compute the same function, this question is again undecidable. When we must select test points, many times we need to know if different paths compute the same function. Yet this result shows that in general this cannot be effectively computed; intuitive and heuristic methods may have to be used. This also shows the essential difficulty with *symbolic evaluation*, since the problem of showing the equivalence of two symbolic expressions corresponding to two control paths is undecidable. Richardson and Clarke (1981) have devised a test strategy that compares symbolic expressions for control paths with symbolic expressions from program specifications; our basic result shows that again the question of whether the expressions are equivalent is undecidable. These authors report reasonable success with this task, however, as well as when expressions are from control paths from the same program. This illustrates an important point: Although a problem may in general be undecidable, techniques can be devised that are effective most of the time. Another approach is to identify special cases in which the problem may be decidable and an algorithmic approach would be appropriate.

2.2.2 Test Selection Problems

It has been proved that the test selection problem is so difficult that no general testing strategy can be constructed. From the observations we have made thus far, this result is not surprising. Howden (1976) has proved the test selection problem to be undecidable. He first makes an observation based upon a result reported by Goodenough and Gerhart (1975), that there exists a finite test set that reliably determines the correctness of a given program over its entire input domain. Howden then goes on to prove that the problem of constructing such a test set for an arbitrary program is undecidable. The essential issue is that, although we know that such a reliable test set exists for each program, no algorithmic method exists for constructing such a set for an arbitrary program. This pessimistic view of the difficulty of the test selection problem is entirely consistent with the observations we have made thus far.

2.2.3 Reachability Questions

Next consider the problem of determining the *feasibility* of a control path for a given program. We have already observed that data flow analysis will determine that a particular control path traverses the control flow graph from the entry node to the terminal node, and that all the variables encountered along that control path are properly defined—this is an example of static analysis. Yet this does not prove that there exists any input data point that will actually execute this control path. Recall that there is a path condition associated with each control path; Davis (1973) has shown that the problem of determining whether a solution exists to a system of inequalities is undecidable. Thus the path feasibility problem is undecidable, since an input point must be found to satisfy the associated path condition. It is interesting to note that, if the predicate constraints in the path condition can all be shown to be linear in the input variables, then the feasibility problem becomes decidable, since the technique of linear programming can be brought to bear on the solution of the linear constraints.

There are a number of testing issues whose decidability properties are quite unexpected. Consider the following problems, given an arbitrary program:

1. Will a given statement ever be exercised by any input data point?
2. Will a given branch ever be exercised by any input data point (where a *branch* corresponds to an arc of the control flow graph)?
3. Will a given control path ever be exercised by any input data point?

4. Will every statement in the program be exercised by some input data?
5. Will every branch in the program be exercised by some input data?
6. Will every control path in the program be exercised by some input data?

Weyuker (1979) has proved all of these problems to be undecidable. This is unexpected because, if we rephrase each question and ask about traversal of each construct from the entry node in the control flow graph, there is an efficient algorithmic solution to each question. Again, the requirement of the exercise of each construct by input data makes the problem undecidable. Weyuker indicates that problems 1 through 6 are extremely important, since the two commonly accepted criteria for adequate testing are that either every statement or every branch in the control flow graph should be executed at least once during the test-data set execution. Furthermore, if we can show that statements, branches, or control paths are never executed, not only can the testing procedures be simplified by recognizing this fact, but the unreachable ("dead") code can be eliminated, thus considerably simplifying the program. The problem is that these apparently static issues are really undecidable, because they must be viewed in a dynamic context as well.

Thus, as we observed earlier, data flow analysis methods can be applied to the control flow graph to identify parts of the control flow graph that cannot be reached by control paths, incomplete control paths can be found, and variables can be detected that are not properly initiated or defined. Yet only some cases of "dead code" can be identified in this way. Code may be unreachable, control paths infeasible, or branches never traversed, and the problems of detecting these situations are in general undecidable. One must keep these facts in mind when designing test data.

2.3 A Theoretical Foundation for Testing

Most researchers would acknowledge the pioneering paper of Goodenough and Gerhart (1975) as the first published paper to establish a sound theoretical basis for testing. Many definitions in testing that are commonly used today emerged from that analysis. Goodenough and Gerhart's "fundamental theorem of testing" has provided a model for other researchers to use as a goal in formalizing a testing concept. The paper on the reliability of path analysis strategies by Howden (1976) further elaborates on the Goodenough and Gerhart results, and many of the concepts in the Howden paper are now commonly used by the testing research community.

2.3.1 Test Selection Criteria

In the paper by Goodenough and Gerhart, the fundamental theorem of testing certainly establishes that there always exists a finite test set that reliably determines the correctness of a given program over its entire input domain. This theorem provides even further insight, however. Goodenough and Gerhart define a "test selection criterion," which specifies conditions that must be satisfied by a finite test set. For example, a criterion for a numerical program whose input domain is the integers might specify that each test should contain a positive integer, one negative integer, and zero; thus $\{-5, 0, 12\}$ and $\{-1, 0, 8\}$ are two of the test sets selected by this criterion.

A test set T is *successful* on program P if P is correct for every element of T. Suppose that T_1 and T_2 are two test sets that satisfy a test selection criterion S. If the criterion S is *consistent*, then T_1 is successful if T_2 is also successful. Suppose that P is incorrect for some input. Then a test selection criterion S is *complete* if there is an unsuccessful test T that satisfies the criterion S. From these definitions, the fundamental theorem of testing of Goodenough and Gerhart, as slightly modified by Howden (1976), follows:

> If there exists a consistent and complete selection criterion S for a program P, and if a test set T satisfying criterion S is successful, then P is correct.

The problem is that in selecting a test selection criterion there is a trade-off between the property of consistency and that of completeness. It is easy to define a test criterion with one property or the other, but it is difficult to achieve both properties. We will cite several examples from Weyuker and Ostrand (1980) that illustrate this point. Assume a program P that computes $X * X$, for X with integer values, while the output specification is $P(X) = X + X$. Since P is correct for $X = 0$ and $X = 2$, and incorrect for all other inputs, a criterion that selects as tests only subsets of $\{0, 2\}$ is consistent but not complete, as it does not indicate the error in P. A criterion that selects subsets of $\{0, 1, 2, 3, 4\}$ exposes the error in P, and is therefore complete but is not consistent, since $T_1 = \{0, 2\}$ is successful whereas $T_2 = \{0, 1\}$ is not. A slight change in the program, while retaining the same output specification, may completely change the completeness and consistency of this criterion. Suppose P' computes $(X + 2)$; then $X = 2$ becomes the only input for which a correct answer is produced. Now we find that subsets of $\{0, 2\}$ become complete but not a consistent criterion for P'. If P' computes $(X + 5)$, the criterion that selects subsets of $\{0, 1, 2, 3, 4\}$ now determines correctness for P'.

2.3.2 Construction of Finite Test Sets

Howden (1976) proves the problem of constructing a reliable finite test set for arbitrary programs to be undecidable. Thus, although we know that a reliable finite test exists, it is the problem of constructing such a set that is undecidable.

In the rest of his paper, Howden defines various types of errors we have already discussed in Section 1.4: computation errors, domain errors and missing path errors. These errors are analyzed by comparing the given program to a hypothetically correct program, which differs from the given program by the error under study. Although this model was quite restrictive, it yielded useful insight into the errors considered.

More recently a fundamental paper by Gourlay (1983) has formally explored the problem of testing when both the specifications and program must be continually modified. This has led to a characterization of the "neighborhood" of a correct program, explored by several other authors, including Howden (1976).

We have briefly examined the landmark papers of Goodenough and Gerhart and of Howden. Most researchers in program testing agree with Goodenough and Gerhart that what testing still lacks is a theoretically sound but practical definition of what constitutes an adequate test. Yet these two papers have certainly established a firm basis for the theory of testing.

3. Static Data Flow Analysis and Testing

Data flow analysis is concerned with program variables, classifying each variable occurrence as a *definition* or a *use*, as defined in Hecht (1977). These techniques are called *static* because they do not require actual execution of the software system, but only analyze the control flow characteristics of the program, together with the behavior of the program variables. The algorithms for data flow analysis are well known and are efficient, making this approach relatively inexpensive, and it can easily be incorporated into existing compilers. This provides an early detection of errors, and certain errors can be detected very reliably. Given the simplicity and economy of this approach, it makes sense that this is done as a preprocessing step, along with the detection of syntax errors, before any dynamic testing approaches are applied.

3.1 Variable Anomalies

In Section 2.1 we observed that a control flow graph is not well formed if there exists no control path by which to reach some specified set of code; this can be ascertained by static analysis. In the same manner, *variable anomalies* can be established by static algorithms; *variable anomalies* consist of undefined variable references, unused variables, or other misuses of program variables. A program variable reference must be preceded by a definition, and without any intervening undefinition (such as an exit from the procedure or scope of that variable). Similarly, a definition must be followed by a reference before another definition or undefinition of that variable. Violation of these rules for variable use does not imply that the program is necessarily incorrect, but it is a tipoff to poor design, and that execution of the program may produce incorrect results. A number of authors have studied these variable anomalies; for example, see Taylor and Osterweil (1980).

3.2 Data Flow Analysis

The notation for this section is drawn from Weyuker (1984) and reprinted by permission of North–Holland Publishing Co., copyright © 1984. A *definition* of variable x is provided either through a READ statement or when x occurs on the left side of an assignment statement. A *predicate use* (or *P-use*) of x occurs when x occurs in a control flow predicate. A *computation-use* (or *C-use*) of x occurs when x is used on the right side of an assignment statement or as an output variable. Of course, a C-use may indirectly affect the flow of control through the program as well.

Recall that in a control flow graph the nodes correspond to *blocks* of statements, always executed as a unit. Thus we can refer to a node containing a reference or a C-use of a particular variable. Similarly, if variables $x_1, x_2, \ldots, x_n$ occur in a predicate in node i, and the two successors of node i are nodes j and k, then we will say that arcs (i, j) and (i, k) contain P-uses of variables $x_1, x_2, \ldots, x_n$.

Recall that our objective here is to ascertain that a variable reference is preceded by a unique definition of that variable; we will trace the flow of control between nodes in a formal and careful way, again using the notation of Weyuker (1984).

Given a control graph and variable x, a path $(i, n_1, n_2, \ldots, n_m, j)$, $m > 0$, containing no definitions of x in nodes $n_1, \ldots, n_m$ is called a *DEF-clear path* with respect to x from node i to node j and from node i to arc (n_m, j). A node i has a *global definition* of a variable x if it has a definition of x and there is a DEF-clear path from node i to some node containing a C-use or arc containing a P-use of x.

There have been a number of systems described in the literature for the efficient determination of DEF-clear paths and global definitions of variables. These systems detect data flow anomalies and inform the user, who can then track down the corresponding program faults quite efficiently. DAVE was a system developed by Fosdick and Osterweil (1976), further described in Osterweil and Fosdick (1978), and with more recent analysis provided in articles such as Osterwell, *et al.* (1981). Frankl and Weyuker (1985) have described their system ASSET for this purpose, as well as for dynamic testing using criteria derived by data flow analysis; these will be described in Section 4.3.

3.3 Critique of Static Analysis

Static analysis can demonstrate that certain kinds of errors are absent from a given program, such as variable anomalies and control flow graphs that are not well formed, and this can be accomplished very efficiently and at low cost. Automatic systems are commonly available for this, which require no expensive programmer or tester interaction and no program execution. Very useful documentation can be generated at this stage, assuring both software developer and user of the reliability of the software to this extent.

Static testing cannot distinguish between execution paths and static control paths, as we have already observed. Static analysis cannot determine and analyze the functionality of a program, for this can only be demonstrated in a dynamic execution mode. Static testing can efficiently eliminate many types of errors, including those associated with variable anomalies, but cannot be effective against many errors only revealed by execution of the program.

4. Coverage Measures

In Section 1.1.2 we gave several coverage measures as examples of white box testing. We want to expand on that discussion to show that, although coverage measures seem to be a systematic approach to testing, many errors will still escape detection.

4.1 Logic Coverage Measures

Sections 4.1.1 through 4.1.4 will present coverage measures referred to as "logic coverage" by Myers (1979), who provides an expanded discussion of this concept. The material in Section 4.1 is reprinted by permission of John Wiley & Sons, Inc., copyright ©1979. There is a certain ad hoc aspect to these measures, illustrated as we move from one coverage criterion to another in order to slightly strengthen it by including a test for an error condition missed in the previous criterion. Also in Section 2.2, we showed that it may not be possible to achieve one-hundred-percent coverage with any of these measures.

4.1.1 Statement Coverage

With *statement coverage*, every statement in the program is to be executed by the test set at least once. Unless one encounters reachability problems, this is certainly a requisite of a test plan. It is not nearly strong enough, however, for consider the example statement

$$\text{IF} \quad X > 0 \qquad \text{THEN} \quad S;$$

In this case, we assume a null ELSE statement; thus with statement coverage, the ELSE condition might never be checked and yet contribute to a serious error.

4.1.2 Branch Testing (Decision Coverage)

We define *branch testing* (or *decision coverage*) if each predicate decision assumes a true and a false outcome at least once during the test set execution (or each possible outcome for a CASE statement). This coverage criterion clearly overcomes the problem with the null ELSE example previously given, but there are problems with decision coverage as well.

For example, consider a program with two successive IF–THEN–ELSE constructs; if tests are selected which execute the THEN–THEN alternative of these predicates, as well as the ELSE–ELSE alternative, then this criterion is satisfied. Yet the THEN–ELSE alternative is not adequately tested, and might well be in error.

4.1.3 Condition Coverage

Another weakness in branch testing is encountered with compound predicates such as

$$\text{IF} \quad (A > 0) \quad \text{AND} \quad (B < 5)$$

Branch testing will treat this compound predicate the same as a simple predicate, testing only for true and false outcomes and ignoring the fact that a false outcome could occur from two distinct Boolean clauses. For this reason, *condition coverage* will require that, during test set execution, each condition in a compound predicate assumes all possible outcomes at least once. Although it appears to strengthen decision coverage, there is still no requirement of a true predicate when both conditions are true.

4.1.4 Multiple Condition Coverage

As a result of this painful evolution of coverage measures, consider *multiple condition coverage*, which requires that during the test set execution all possible combinations of condition outcomes in each predicate are invoked at least once. It should be clear that this coverage measure implies (or is stronger than) decision coverage and condition coverage criteria. The reason for the term "possible" in the definition is that some combinations may be redundant. For example, for

$$(X \leq 5) \quad \text{AND} \quad (X < 10)$$

there are only three (not four) conditions generated, $X \leq 5$, $5 < X < 10$, and $X \geq 10$.

There are obvious deficiencies with any coverage measure of this type if used alone for test generation. These measures do not guarantee that every path is tested, nor are they based on a sufficient theoretical base to make clear which paths need not be tested. More to the point, these coverage measures do not take into account interaction between different predicates, which might occur along a common path. As a matter of fact, there have been proposals to take pairs of (or multiple) predicates at a time and generate more complex coverage measures. In the limit, as all possible combinations of predicates are considered, this is equivalent to considering all possible paths in test generation.

4.2 Iteration Loops

In the discussion of coverage measures we did not explicitly indicate how iteration loops would be treated as part of the coverage measure. One approach sometimes used is that an iteration loop is executed at least once or not at all as a portion of the coverage measure. Another approach is that coverage of the iteration loop will require execution exactly once (assuming this is possible), at least one execution multiple times, and no execution of the iteration loop as an adequate coverage for test generation. Clearly these ad hoc measures do not take adequate account of the structural information contained within the iteration loop, and are totally unsatisfactory. Fortunately there are some results by Zeil (1981) and by Wiszniewski (1985) that give some more helpful indications of the number of executions of an iteration loop for test generation, and these are based upon a more solid theoretical framework. These approaches will be discussed in Section 6.3.5.

4.3 Data Flow Testing

A number of authors have recommended the use of data flow analysis as the basis for dynamic testing; for example, see Woodward *et al.* (1980), Laski and Korel (1983), Rapps and Weyuker (1985), Weyuker (1984), and Ntafos (1984), with his description of "required element testing." These methods can be viewed as generalizations of the logic coverage measures, especially as developed in Weyuker (1984).

Following the development and notation from Weyuker (1984), with permission of North–Holland Publishing Co., copyright © 1984, and continuing definitions from Section 3.2, we can obtain a family of data-flow testing criteria, using $\mathscr{P}$ as a set of paths through the control flow graph:

1. $\mathscr{P}$ satisfies the *all-definitions* criterion if every global definition is used.

2. $\mathscr{P}$ satisfies the *all-uses* criterion if $\mathscr{P}$ includes a path from every global definition to each of its uses.
3. A *simple path* is one in which all nodes, except possibly the first and last, are distinct. A *loop-free path* is one in which all nodes are distinct. A path $(n_1, \ldots, n_j, n_k)$ is a *DU-path* with respect to a variable x if n_1 has a global definition of x and either (i) n_k has a C-use of x and $(n_1, \ldots, n_j, n_k)$ is a DEF-clear simple path with respect to x, or (ii) (n_j, n_k) has a P-use of x and $(n_1, \ldots, n_j)$ is a DEF-clear, loop-free path with respect to x.

Then $\mathscr{P}$ satisfies the *all-DU-paths* criterion if for every node i and every x that has a global definition in i, $\mathscr{P}$ includes every DU-path with respect to x. Note that if there are multiple DU-paths from a global definition to a given use, they must all be included in paths of $\mathscr{P}$.

Figure 3 shows a family of data-flow criteria from Weyuker (1984); similar hierarchies have been developed by Korel and Laski (1985) and Clarke *et al.* (1985). In Figure 3 the *all-nodes* criterion corresponds to statement coverage, and the *all-edges* criterion corresponds to branch coverage. The arrows in Figure 3 imply strict inclusion, indicating that the higher-level criterion implies a lower-level criterion.

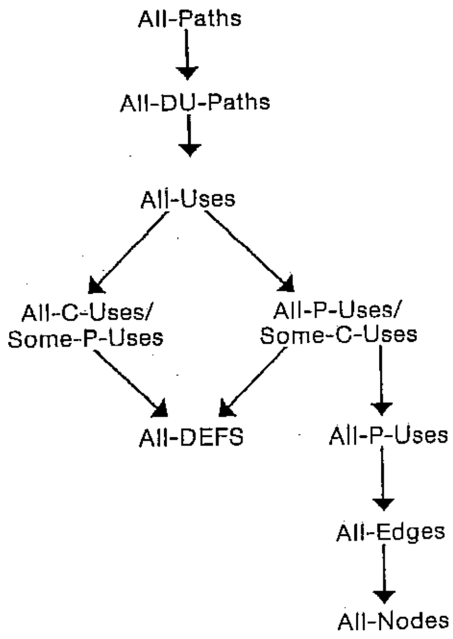

FIG. 3. Family of data-flow test-selection criteria. (From Weyuker, copyright © 1984, North–Holland Publishing Co.).

Moreover, Weyuker (1984) has also analyzed the complexity of these criteria. Consider a program with m variables, a assignments, b input statements, and c conditional transfers. If d represents the number of definitions in the program, it follows that

$$(a + b) < d < a + (b \cdot m)$$

since each input statement defines between one and m variables.

The all-nodes and all-edges criteria require at most $(c + 1)$ test cases. The all-DEFs criterion requires at most $(a + b \cdot m)$ test cases. All criteria indicating uses require at most

$$(1/4)(c^2 + 4c + 3)$$

test cases (see the derivation in Weyuker, 1984). The all-DU-paths criterion requires at most 2^c test cases, whereas it is well known that the all-paths criterion requires exponential effort. Weyuker argues that only extreme program configurations would require exponential test cases for all-DU-paths, and that it constitutes a reasonable criterion for most practical situations.

5. Mutation Analysis and Functional Testing

In this section we review *mutation analysis* and a variant of this approach called *weak mutation testing*. As these techniques have developed, they now appear to be related to the classic black box approach, functional testing. Functional testing will also be examined, and related to mutation testing.

The primary objective of *mutation analysis* is to evaluate the degree to which a test set exercises a program, rather than the initial generation of that test. This approach was proposed by Budd *et al.* (1978), and a considerable literature exists on various developments and experiments using this approach; for example, a substantial development is given in DeMillo *et al.* (1978) and in Budd (1981). The area has been associated with a degree of controversy; this methodology is now well accepted as one systematic approach to testing, and it is especially valued in that a substantial amount of experimental work has been conducted and reported. Much of the material in Section 5.1 is drawn from Budd (1981), reprinted by permission of North–Holland Publishing Co., copyright ©1981.

Since the goal of testing is to increase our confidence in the program being tested, good test data should be associated with the form and function of the program. Mutation analysis develops a systematic approach by producing a large number of small incremental modifications in individual program

statements. Thus the classes of faults in a program will be explicitly defined and associated with these modifications. Mutation analysis can be viewed as providing a measure of test data quality, and there have been a number of proposals to utilize this analysis in an iterative mode to improve the given test set.

A number of authors have suggested methods for functional testing, and there are also a substantial number of systems based on this approach. The fundamental idea is that functions be identified within the computer software or elsewhere, and in order to test the program, each of these functions must be tested over appropriately selected test cases. We shall see that the problem is to approach the generation of the functions and test cases systematically, and eventually automatically.

5.1 What is Mutation Analysis?

Mutation analysis is based on four assumptions; it is probably the defense of these assumptions that has provided a great deal of the controversy associated with this approach:

1. *The competent programmer hypothesis:* Experienced programmers write programs that are either correct or almost correct.
2. *The coupling effect:* Test data that detects small faults in a program (such as small changes in individual statements) is also likely to detect complex faults in that program.
3. *Mutant operators:* An adequate set of predefined incremental operators can be identified to capture nearly all simple faults in a given program; these operators will in general be different for each high-level programming language.
4. *Test oracle availability:* As usual, we must have some systematic way to ascertain the correctness of output for test data submitted to a given program.

Given a program P to be tested and a set of test cases T, T is first executed with P, and we assume this execution produces no errors. A number of alternative programs, called *mutants* of P, are then produced by small changes in P. Test set T is then executed on each of these mutants. Continuing this biological analogy, if at any point P and this mutant produce different output, we say this mutant has *died*; if identical output responses are obtained, we say the mutant *survives*.

If a large number of the mutants survive, then one conclusion that could be drawn is that the test set T is insufficient and additional test cases are needed. If the number of living mutants is small, and we are confident that the set of defined incremental operators, called *mutant operators*, is sufficient, then we can conclude that the test set has been relatively effective in eliminating mutants based on the incremental mutant operators. The test data must have been tied closely to the form and function of the program being tested. Also, the program has been carefully examined in generating the test data, and has been extensively exercised by that data. Budd (1981) has conducted empirical studies which support the assertion that if the set of test cases can eliminate a sizable percentage of mutants for a program P, then it is likely that P is correct.

The surviving mutants provide the tester with a new method of generating test data: to eliminate these mutants. This leads to an iterative testing process, terminating when nearly all the mutants have been eliminated.

5.1.1 Generation of Mutant Programs

Budd (1981) points out that in the generation of mutant programs it is not reasonable to construct the mutants in such a way that each corresponds to a possible fault the programmer might include in a given program. Rather, the mutant operators are chosen as indicators of whether the test data is sensitive to small changes in the program.

In practice, a number of mutation analysis systems have been developed and applied to FORTRAN, ALGOL, and COBOL languages, as well as a pilot system for an assembly language. Budd (1981) indicates that the system works best for languages in which small syntactic changes tend to produce only small semantic changes in the program; thus much higher-level languages such as APL or SETL would not be as appropriate for mutation analysis.

Mutant operators include such incremental changes to a program as changing a + operation to a − operation, adding one to an arithmetic expression, or interchanging two variables. The EXPER system (Budd, 1981) for FORTRAN programs contains twenty-two different types of such operators; in this description Budd shows how this mutation analysis implementation achieves the same testing effects as statement or branch coverage testing, data-flow analysis, predicate testing, and special values testing.

One of the serious problems with mutation analysis is the very large number of mutants that are generated and must be processed, even though many will quickly die off as the test set is applied. This growth has been characterized as $O(L^2)$, where L represents the number of lines in the program, yet the number of mutants may also depend upon the amount of data being processed. For example, Budd (1981) reports that two programs, each thirty-three statements in length, were analyzed by mutation analysis. The first was a text processing program, producing 859 mutants, whereas the second, a complex program for performing 3 × 3 matrix multiplication, had 2,382 mutants. Budd also argues that a more careful complexity analysis shows the number of mutants for an "average" program to vary with the product of the total number of data references (including constants) and the number of distinct data references. Even though this analysis is a bit more optimistic than $O(L^2)$, the number of mutants to consider grows very rapidly for even moderate-sized programs.

5.1.2 The Problem of Equivalent Mutants

A mutant program is said to be *equivalent* to the given program P if both have identical input–output responses. The set of equivalent mutants relative to program P then cannot be distinguished from P by any test sets, and will always survive any test set in mutation analysis. Budd (1981) indicates that between 4% and 10% of generated mutants are equivalent, with heavy clustering at 4%.

As indicated in Section 2.2, it is generally undecidable whether two programs are equivalent, so it will be difficult or impossible to tell whether during mutation analysis each surviving mutant is equivalent, or whether it could be removed by a suitably expanded test set. Researchers such as Budd (1981), working with mutation analysis, have indicated that in many instances simple techniques and intuition can be used to identify surviving equivalent mutants. Still, this remains as a serious theoretical and practical problem in the implementation of mutation analysis.

5.1.3 Summary of Mutation Analysis

Mutation analysis has emerged as providing a unifying approach to testing, both in test-set selection and in determination of the quality of the test set. A number of authors, such as Howden (1982), have referred to *mutation testing* when some of the techniques of mutation analysis are applied to test-set selection. Specifically this refers to the construction of tests designed to distinguish between mutant programs that differ by a single mutation transformation.

The underlying assumptions of mutation analysis are still quite controversial and contentious; the method can still be useful if used carefully and without requiring the validity of all the underlying assumptions. There are still serious problems in the implementation. Even a small number of mutation operators can lead to an enormous number of mutant programs. Another problem is the issue of equivalent mutants, and how surviving mutants can be identified as equivalent to the given program or not. The method requires substantial and complex software support, and there is not a systematic or automatic method for the generation of additional test data to eliminate surviving mutants.

5.2 Weak Mutation Testing—Another Approach

Howden (1982) has suggested a testing method which has many apparent similarities to mutation testing and hence has been called *weak mutation testing*. The description of this technique is drawn from that publication, reprinted by permission of IEEE, copyright © 1982. Test selection rules are obtained from arithmetic expressions and relations. Weak mutation testing differs philosophically from mutation testing in several ways:

1. In mutation testing, functions computed by the entire program are tested and compared to distinguish mutants. In weak mutation testing the focus is upon the testing of statement-level functions and expressions.
2. In mutation analysis, to obtain a comprehensive set of mutant programs the mutant operators must by necessity be quite dependent upon the specific programming language used. We shall see that, although weak mutation analysis concentrates on functions at the statement level, the defined mutations are not as dependent upon the specific programming language.
3. Howden (1985) points out that in mutation testing the classes of faults for which it is effective are explicitly defined; there is, however, no direct global way to obtain tests that will reveal these classes of faults. There is a trade-off here, in which the tests can be obtained only if the fault detection capability is *weakened*. This accounts for the motivation behind the term "weak mutation testing," where local test criteria can be obtained at the price of perceiving the effect of the weak mutation on the overall program behavior. Thus we shall see that we may have lost the ability to obtain global test sets that produce the required local test criteria to detect weak mutations.

5.2.1 Component Testing

Howden (1982) defines a *component* as an elementary computational structure in a program; examples of components are references to variables, arithmetic expressions and relations, and Boolean expressions. If P is a program containing a component C, then there is a mutation transformation that can be applied to C to produce C'; P' is then the mutant program corresponding to P and containing C'. In weak mutation testing it is required that a test t be constructed in which C is executed as t is applied to P, and that in at least one such execution of C, C produces a different value from C'. Notice that, even though C' produces a different value from C under test t, it is possible for programs P and P' to compute the same output under test t.

The component mutations then consist of the following:

1. *Variable reference:* This component mutation causes the component to reference a different variable. If v is a variable associated with a component C, then in order for C to compute a value different from a possible mutation C' of C, it is necessary to execute C in an environment in which v has a value different from the values of all other variables in that environment.
2. *Variable assignment:* This component mutation causes the component to assign the value to a different variable. If v is a variable to which C assigns a value, then in order for C to return output different from a possible mutation C' of C, it is sufficient to execute C over data in which the value stored by C into v is different from the value currently stored in v.

These two types of component mutations are primitive and appear as parts of the other three kinds of components:

3. *Arithmetic expression:* This component mutation consists of arithmetic expressions that are off by an additive constant, off by a multiplicative constant, and have incorrect coefficients. The first two types of mutations can be easily distinguished from corresponding expressions, for if E' is an additive or multiplicative constant mutation of an arithmetic expression E, then it is sufficient to execute E over a single vector of values in order to distinguish E from E'. If E is an arithmetic expression in which one or more coefficients have been changed, then distinguishing mutation arithmetic expressions is more complex, and the size of test sets required to differentiate them may be quite large; Howden (1982) gives some ideas as to how these test sets can be chosen.
4. *Arithmetic relation:* This component mutation consists of arithmetic relations that contain incorrect relational operators and off-by-an-additive constants. An arithmetic relation R ($\exp_1 R \exp_2$) can be distinguished from a mutation R' in which the relation has been changed by executing R over data for which $\exp_1 < \exp_2$, $\exp_1 = \exp_2$, and $\exp_1 > \exp_2$. The off-by-a-constant mutation for an arithmetic relation can be detected by a suitably chosen single test point.

Howden (1982) indicates that his results for arithmetic relation mutations have been derived from work described by Foster (1980). Foster based his development on the assumption of integer variables, but the results can easily be generalized to real numbers. In his paper Foster has contributed several other ideas for testing techniques where no model is available for analysis:

5. *Boolean expression:* This component is a function of the form $B(E_1, E_2, \ldots, E_n)$, where E_i, $1 \leq i \leq n$, is an arithmetic expression or variable that evaluates to true or false, where the E_i are transformed by logical operators OR, AND, and NOT. A Boolean expression mutation can

then be viewed as one or more of the expressions E_i with a modified truth value. A test set can be constructed to distinguish B from all other Boolean expressions B' by constructing tests t so that all possible truth values for the E_i are generated. Again, this test set can grow exponentially, but Howden (1982) discusses some ways to control the growth of this set.

5.2.2 Comparison to Other Methods

Howden (1982) argues that weak mutation testing can be viewed as a refinement of branch testing. Weak mutation testing corresponds to a wider class of errors, and there are some subtle errors that require evaluation of branch functions over special kinds of test values, the sort of careful evaluation conducted in weak mutation testing but not in branch testing.

There are several advantages of weak mutation testing over mutation testing. The former is more efficient, as it is not necessary to carry out a separate program execution for each mutation; moreover, as we have seen by the various types of component mutations, only a few (or even one) tests may be required. A second advantage is that test data can be specified a priori, over which a component mutation must be executed in order that a different output value can be obtained. This means the user will have guidance as to what types of tests should be applied to the component. The disadvantage of weak mutation testing is that the overall program may act correctly over this same set of data and not indicate a different output behavior.

5.3 Functional Testing

A testing strategy that has been popular in industrial and commercial software applications has been known as "functional testing." This has traditionally been a black box approach in which the functional properties of the requirements or specifications are identified and test data selected to specifically test each of those functions.

There are two problems with this approach. First, although requirements and specifications provide many meaningful functions that can be the focus for functional testing, software may contain a much richer collection of functions than those put forth in specifications. Moreover, specifications are often inadequately described to provide the detailed information required for testing. A second problem has been the lack of any unifying and fundamental theoretical basis for such testing. Thus the commercial efforts using functional testing are both ad hoc and incomplete in scope.

Howden (1980; 1985) has developed an underlying theory for functional testing and has extended the concept to include functional components of the software as well. In this way, to a great extent he has overcome the two problems identified with the previous concept of functional testing. Howden's philosophy is that it is important to not only test the functions implemented by the program as a whole (which are best captured in requirements or specifications), but also to test the functions that constitute various parts of the program. There are many ways to decompose the program, each giving a different description of the functions that comprise that program. Howden's theory provides an insight into that decomposition, which at best should mirror the synthesis process by which the programmer has constructed the program, beginning with simple and primitive functions and routines, and evolving these into complex systems.

Howden provides two elements in his theory: functional synthesis and testability. Functional synthesis is a view of the way programs are written, and a summary of these ideas is given in the next section. Testability is a positive characteristic of this particular decomposition of the programming process, and is shown to be the essential ingredient for successful functional testing. Much of this material is drawn from Howden (1985), reprinted by permission of IEEE, copyright ©1985.

5.3.1 Functional Synthesis

Four types of functional synthesis are identified by Howden as at the appropriate level for functional testing:

1. *Algebraic synthesis*—The use of algebraic expressions in assignment statements and predicates provides an initial building block. As usual, these expressions are built from variables with either numerical or Boolean values.
2. *Conditional synthesis*—A more complex building block is the familiar IF–THEN–ELSE construction, but note that the predicate, the THEN, and the ELSE clause each constitute functions, where each is built up from algebraic expressions of various forms.
3. *Iterative synthesis*—Loop iteration is another fundamental construct, and the functional manifestation is most easily seen in the WHILE form, where the predicate function determines termination, whereas the body of the loop provides an additional function.
4. *Control synthesis*—The most complex building block of Howden is a high-level view of control using the familiar state transition model. This allows the description and modeling of many qualitative state concepts in programs which cannot be captured numerically; e.g., "user has failed to provide proper input data" or "end-of-file has been reached." The

programmer is expected to know the operation of the program sufficiently well to provide state transition diagrams as models of various important functional parts of that program.

5.3.2 *Testability*

Howden (1985) defines *testability* as the property that a finite set of tests can be specified that will determine if the program that implements a function contains one of a specified set of faults. More specifically, let f be a function implemented by all or part of a program, and let f' be the hypothetically correct but unknown function. Suppose a set of functions F is known to contain both f and f', and that it is possible to construct a finite set of tests T such that, for any function f' in F, if $f = f'$ over all tests in T, then $f = f'$ over all input data. Then f is said to be *testable* relative to F. The concept of *testability* is a further development of the notion of *completeness criteria*, identified earlier by Howden (1981; 1982).

As an example of a testable algebraic expression that might occur in algebraic synthesis, consider the form

$$f(x, y) = (ax + b)/(cy + d)$$

Assume that F contains f and all other functions that differ by only a single parameter from the set of four in this function. Then f is testable relative to F, for a single nonzero value of x and y for which the denominator is nonzero will suffice as set T.

At this point notice the similarity between testability and the test selection rules given in Section 5.2.1 for Howden's weak mutation testing. Howden (1982) developed the concept of testability and presented it along with the weak mutation testing approach. Although both are shown to be testable by Howden's criteria, they do represent different approaches to testing.

The concept of control synthesis differs substantially from the weak mutation components described in Section 5.2.1. Howden indicates that he was motivated to include control synthesis in functional testing as a result of an empirical study of errors in a large COBOL data processing system. This study showed that control synthesis faults had to be addressed using a different functional model.

In Section 1.5 we discussed the need for a test oracle, which is the type required for program parts for testing algebraic, conditional, and iterative synthesis. Howden (1982) requires a different type of oracle for control synthesis; the oracle (or programmer) must know what sequence of functions should be performed for a certain test case. Thus control synthesis can detect missing computational faults, as illustrated in Figure 4. Howden argues that, if the only kind of fault possible is a missing computation fault and we are provided with a control synthesis oracle, then $f = f_1 f_2$ is testable for the missing computation fault f_3. When a systematic execution of the program is conducted, the required functional sequence will not be executed, thus revealing the missing computation fault.

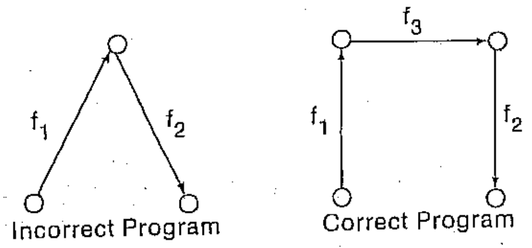

FIG. 4. Missing computational fault. (From Howden, copyright © 1985, IEEE).

5.3.3 *Function Identification*

Howden (1980) provides the sources of functions; they should be selected from

1. specifications—if informal, then verbs provide the functions; if formal, the functions can be obtained from formal assertions, tables, and formulas;
2. programs—select functions from elementary program statements, subroutines, and subpaths;
3. design information and design documents—these materials provide an additional source of functions; it is ideal if a mapping can be found from design functions to specific code fragments.

Howden emphasizes that the program should be well understood before functional testing is used. This advice certainly applies to other methods of testing as well.

5.3.4 *Functional Test Coverage*

Given the sources of functions identified in Section 5.3.3, various means must be found to select tests for these functions. Howden (1980) has identified the following methods of test coverage, and has found them to be useful for functional testing:

- tests to span domains of output variables or expressions, as well as input variables

- tests for both extreme values and interior values of synthesis functions
- tests for illegal values of variables or synthesis functions
- tests for various array or vector patterns; this may require partitioning arrays in various ways to provide the important test sets.

Howden (1980) performed experiments over a set of statistical and numerical analysis programs from the IMSL package (1978), and these programs contained eight distinct errors. He found that static analysis was most effective for 41 errors and dynamic testing for 42 of the errors. Functional testing was compared to a structural testing approach, consisting of a combination of branch testing and path testing. Functional testing was most effective in detecting 31 of the errors, whereas the structural testing approach was most effective in detecting ten of these errors. Howden (1980) has also applied the functional testing method to data processing programs in COBOL, with similar success.

5.3.5 *Functional Testing and Coverage Measures*

Howden (1980) observes that branch testing can be interpreted as an approximation to functional testing. It does not force combinations of branches, and does not construct testing over functionally important cases. We presented in Section 4.3 Weyuker's arguments that data flow coverage is a more powerful approach than branch testing. Howden indicates that if two program parts are related by data flow, then they may comprise the same function, and thus he argues that data flow methods can be viewed as functional testing.

5.3.6 *Functional Testing and Mutation Testing*

We have already observed the intimate relationship between functional testing and weak mutation testing. In mutation testing, entire mutant programs are considered as functions; whole programs are constructed using mutant operators. Thus there is no building up of intermediate synthesis, as is done in software design. Howden (1985) also observes that, as a result, mutation testing is not testable in the sense defined in Section 5.3.2.

6. Path-Oriented Testing Models

Path-oriented testing is based on the use of the control flow of the program. In Section 1.1.2 we indicated that path-oriented testing requires that a set of paths be selected in some manner, and then subsequently a method must be found to select input test data that will cause those paths to be executed. In this section we will provide some examples of methods to accomplish both of these goals; as with many other aspects of testing we have discussed, both of these problems are difficult.

Despite the fact that we have identified the selection of paths and the selection of input data to execute those paths as separate operations, one of the conceptual advantages of path-oriented testing is the explicit association of the path and the input data that causes its execution.

Because of the presence of iteration loops, there is potentially an infinite number of distinct paths in a program. Even in a program without iteration loops, there is potentially an exponential number of distinct paths, as a function of the number of predicates in the program. Thus, for any nontrivial program it does not make sense to generate test data for all paths in that program.

In Section 2.1.1 a path was identified as infeasible if no input data exists that causes that path to be executed. We indicated that the problem of determining the feasibility of a path is undecidable; the problem of identifying infeasible paths is difficult at best in a practical situation. Infeasible paths are not indicative of improper program design and coding, but will occur in well-designed software. The tester must be aware of this problem.

We will consider how program paths can be used to guide the testing process, and some of the factors that must be accounted for to accommodate testing. The path selection process will be presented, including some metrics that are also available for this selection.

Three different path-oriented approaches will be presented: domain testing, perturbation testing, and partition testing. Another technique, called the sufficient paths criterion, will provide some insight into how paths might be selected; some experimental work conducted using this technique will be reported. An overview of domain testing and the sufficient paths criterion is given in White *et al.* (1981).

6.1 Path Testing Approaches

When paths are used for testing, this allows various graph-theoretic models to be used to assist in the path selection. For example, in order to achieve either statement coverage or branch coverage, a set of paths may be defined as a graph covering problem, where the paths cover either the vertices or the arcs, respectively, of the control flow graph for a given program. In order to present this approach more precisely, let us specifically consider branch coverage, which we know to be a stronger testing approach than statement coverage.

Huang (1979) defines a *minimal covering set* of paths and the associated path conditions such that if these path conditions are all satisfied, then every arc in the control flow graph will be traversed at least once (corresponding to branch coverage); moreover, this set should be minimal with respect to this property. A test set is then *minimally thorough* if every arc (or branch) in the control flow graph is traversed at least once during the execution of the test set; moreover, this test set should be minimal with respect to this property. From the discussion in Section 2.2, recall that if we find the minimal covering set of paths, this does not guarantee a minimally thorough test, as one or more of the paths in the minimal set may be infeasible. Ntafos and Hakimi (1979) have provided an efficient polynomial solution to the minimal covering path set problem; the complexity is $O(W^2)$, where W is the number of vertices of the control flow graph.

One would like to have a metric in order to guide path-oriented testing; for example, software managers are infamous in their desire for a single number to measure how testing is going—i.e., "test coverage." A study of metrics has been given by Woodward *et al.* (1980), including metrics for statement coverage, branch coverage, and some generalizations; the authors call these "test effectiveness ratios," measuring the percent coverage of the total number of required structures. In addition, Woodward *et al.* (1980) have provided some important experimentation on a collection of numerical software, in order to gain experience with these metrics. As a result of these experiments they have provided some further insights into path-oriented testing. They advocate testing as many of the shorter feasible paths as possible, since they are simpler and yet achieve good coverage. They further note that the presence of infeasible paths impedes the achievement of 100% coverage measures. They have provided a further study of how the number of infeasible paths can be reduced by utilizing a systematic methodology; this research is continuing. They also add a warning which we should make sure is communicated to software managers: An achievement of 100% coverage metric does not provide any guarantee of the absence of errors in the software. Other, more sophisticated metrics can be based on the data flow testing hierarchy of Weyuker (1984), which we discussed in Section 4.3 and presented as Figure 3. Experimentation with these metrics will be reported in the research literature over the next several years.

For the problem of the termination of path-oriented testing, another alternative to coverage metrics is to phrase the problem another way: In the selection of paths for testing, is there a point at which subsequent paths will give little or no information, and we can stop? This approach has led to the "sufficient paths criterion," which will be discussed in Section 6.3. This approach currently has a number of serious limitations, but does represent an interesting alternative to coverage metrics, especially in that this sufficient paths criterion can lead to the selection of improved paths for testing.

An excellent discussion of path-oriented testing and the selection of paths for this purpose is provided in the monograph *Software Testing Techniques* by Boris Beizer (1983). His approach is quite pragmatic, and he possesses a strong base of practical experience, which pervades his discussions. Beizer recommends the selection of simplest and functionally sensible paths to achieve the target coverage metric. He also suggests that additional paths should differ in small variations from previous paths selected; he argues that paths selected for testing represent an experiment, and in experimental design one attempts to change as few variables as possible in subsequent experiments. Beizer argues for paths that execute iteration loops once, more than once, and no times (if possible). He indicates that Huang (1979) has shown that some initialization problems can only be detected by two or more passes through the iteration loop, and this is his rationale for requiring multiple executions of an iteration loop within some selected path. Beizer has many other practical suggestions for the selection of test paths and should be consulted for other ideas.

Finally, one of the advantages of path-oriented testing is that these methods tend to be easier to automate than other approaches, such as functional testing. This can be very important, as testing requires extensive involvement and time of an experienced tester; a software producer becomes quite vulnerable if an experienced testing professional should leave his organization.

6.2 Domain Testing

The objective of domain testing is to demonstrate that it is possible to select test data for a restricted set of programs to detect a specified type of error, and yet to characterize the extent to which these errors could be detected for that program class. Much of the material in this section is drawn from White *et al.* (1981) and is reprinted by permission of North–Holland Publishing Co., copyright © 1981. In the following development, concepts of predicate interpretations and input space structure will be presented before the domain strategy is discussed.

6.2.1 *Predicate Interpretation*

A simple predicate is said to be *linear* in variables $V_1, V_2, \ldots, V_N$ if it is of the form

$$A_1V_1 + A_2V_2 + \cdots + A_NV_N \text{ ROP } K$$

where K and the A_i are constants and ROP represents one of the relational operations ($<, >, =, \leq, \geq, \neq$).

In general, predicates can be expressed in terms of both program variables and input variables. In generating input data to satisfy the path condition, however, we must work with constraints in terms of only input variables. If we replace each program variable appearing in the predicate by its symbolic value calculated in terms of input variables along that path, we get an equivalent constraint called the *predicate interpretation*. A single predicate can appear on many different execution paths. Since each of these paths will in general consist of a different sequence of assignment statements, a single predicate can have many different interpretations. The following program segment provides example predicates and interpretations:

```
READ $A,B$;
IF $A > B$
   THEN $C = B + 1$;
   ELSE $C = B - 1$;
$D = 2 * A + B$;
IF $C \leq 0$
   THEN $E = 0$;
   ELSE
         DO $I = 1,B$;
               $E = E + 2 * I$;
         END
IF $D = 2$
   THEN $F = E + A$;
   ELSE $F = E - A$;
WRITE $F$;
```

In the first predicate, $A > B$, both A and B are input variables, so there is only one interpretation. The second predicate, $C \leq 0$, will have two interpretations, depending on which branch was taken in the first IF construct. For paths on which the THEN $C = B + 1$ clause is executed, the interpretation is $B + 1 \leq 0$, or equivalently, $B \leq -1$. When the ELSE $C = B - 1$ branch is taken, the interpretation is $B - 1 \leq 0$, or equivalently, $B \leq 1$. Within the second IF–THEN–ELSE clause, a nested DO loop appears. The DO loop is executed

$$\text{no times if } B < 1$$

$$\text{once if } 1 \leq B < 2$$

$$\text{twice if } 2 \leq B < 3$$

$$\text{etc.}$$

Thus the selection of a path will require a specification of the number of times that the DO loop is executed, and a corresponding predicate is applied to select the input points that will follow that particular path. Even though the third predicate, $D = 2$, appears on four different paths, it has only one interpretation, $2 * A + B = 2$, since D is assigned the value $2 * A + B$ in the same statement for each of the four paths.

6.2.2 Input Space Structure

An *input space domain* is defined as a set of input data points satisfying a path condition, consisting of a conjunction of predicates along the path. For simplicity in this discussion, each of these predicates is assumed to be simple. The input space is partitioned into a set of domains. Each domain corresponds to a particular execution path in the program and consists of the input data points that cause the path to be executed.

The boundary of each domain is determined by the predicates in the path condition and consists of *border segments*, where each border segment is the section of the boundary determined by a single simple predicate in the path condition. A "redundant" predicate is implied by some subset of the other predicates of the path condition; it can be removed, as no border segment will correspond to a redundant predicate. Each border segment can be open or closed, depending on the relational operator in the predicate. A *closed border segment* is actually part of the domain and is formed by predicates with $\leq$, $\geq$, or $=$ operators. An *open border segment* forms part of the domain boundary but does not constitute part of the domain, and is formed by $<$, $>$, and $\neq$ predicates.

The general form of a simple linear predicate interpretation is

$$A_1X_1 + A_2X_2 + \cdots + A_NX_N \text{ ROP } K$$

where ROP is the relational operator, X_i are input variables, and A_i, K are constants. The border segment defined by any of these predicates is, however, a section of the surface defined by the equality

$$A_1X_1 + A_2X_2 + \cdots + A_NX_N = K$$

since this is the limiting condition for the points satisfying the predicate. In an N-dimensional space this linear equality defines a hyperplane, which is the N-dimensional generalization of a plane.

Consider a path condition composed of a conjunction of simple predicates. These predicates can be of three basic types: equalities ($=$), inequalities ($<$, $>$, $\leq$, $\geq$), and nonequalities ($\neq$). The use of each of the three types results in a markedly different effect on the domain boundary. Each equality constrains the domain to lie in a particular hyperplane, thus reducing the dimensionality of the domain by one. The set of inequality constraints defines a region within the lower-dimensional space specified by the equality predicates.

Equalities can arise from predicates that occur explicitly in the program, or that form from two or more inequalities that produce a coincidental equality. For example, if $X \leq A$ and $X \geq A$ are two inequalities that occur in a single path condition, then the result is a coincidental equality $X = A$.

6.2.3 Domain Testing Assumptions

The domain testing strategy is designed to detect errors and will be effective in detecting errors in any type of domain border under certain conditions. Test points are generated for each border segment, which, if processed correctly, determine that both the relational operator and the position of the border are correct. An error in the border operator occurs when an incorrect relational operator is used in the corresponding predicate, and an error in the position of the border occurs when one or more incorrect coefficients are computed for the particular predicate interpretation. The strategy is based on a geometrical analysis of the domain boundary and takes advantage of the fact that points on or near the border are most sensitive to domain errors. A number of authors have made this observation—e.g., Boyer *et al.* (1975) and Clarke (1976).

It should be emphasized that the domain strategy does not require that the correct program be given for the selection of test points, since only information obtained from the given program is needed. It will be convenient, however, to be able to refer to a "correct border," although it will not be necessary to have any knowledge about this border. Define the *given border* as that corresponding to the predicate interpretation for the given program being tested, and the *correct border* as the border that would be calculated in some correct program.

There are limitations inherent to any testing strategy, and these also constrain the domain strategy. Two such limitations were defined in Section 1.4 as coincidental correctness and missing path errors. As applied to domain testing, coincidental correctness can occur when a specific test point follows an incorrect path, and yet the output variables coincidentally are the same as if that test point were to follow the correct path. This test would then be of no assistance in the detection of the domain error that caused the control flow change. No path-oriented strategy can circumvent this problem.

The domain testing strategy will be developed and validated under a set of simplifying assumptions:

1. Coincidental correctness does not occur for any test case.
2. A missing-path error is not associated with the path being tested.
3. Each border is produced by a simple predicate.

4. The path corresponding to each adjacent domain computes a different function than the path being tested.
5. The given border is linear, and if it is incorrect, the correct border is also linear.
6. The input space is continuous rather than discrete.

Assumptions 1 and 2 have been shown to be inherent to the testing process, and cannot be entirely eliminated. Recognition of these potential problems, however, can lead to improved testing techniques. Assumptions 3 and 4 considerably simplify the testing strategy, for with them no more than one domain need be examined at one time to select test points. As for the linearity assumption, 5, the domain testing method has been shown to be applicable for nonlinear boundaries, but the number of required test points may become inordinate, and there are complex problems associated with processing nonlinear boundaries in higher dimensions. The continuous input space assumption, 6, is not really a limitation of the proposed testing method, but allows points to be chosen arbitrarily close to the border to be tested. An error analysis for discrete spaces is available (White *et al.*, 1978) and shows that pathological cases do exist in discrete spaces, for which the testing strategy cannot be used, but that these occur only when domain size is on the order of the resolution of the discrete space itself.

Any program that satisfies constraints 1 through 6 will be referred to as a *linearly domained program.*

6.2.4 Test Point Selection

The test points selected will be of two types, defined by their position with respect to the given border. An *ON* test point lies on the given border, while an *OFF* test point is a small distance ϵ from, and lies on the open side of the given border. Therefore we observe that, when testing a closed border, the ON test points are in the domain being tested and each OFF test point is in some adjacent domain.

Figure 5 shows the selection of three test points A, B, and C for a closed inequality border segment. If the OFF test point C is projected down on line segment AB, then the projected point must lie strictly between A and B on this line segment. Also, point C is selected a distance ϵ from the given border segment and will be chosen so that it satisfies all the inequalities defining the domain except for the inequality being tested.

The domain testing strategy for the two-dimensional case can be extended to the general N-dimensional case in a straightforward manner. Since an $(N - 1)$-dimensional hyperplane border segment is determined by N linearly

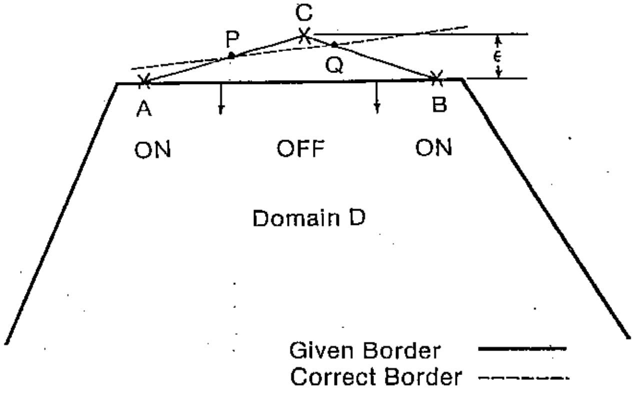

FIG. 5. Test points for a two-dimensional linear border.

independent points, we have to identify N points on the correct border and these points must be guaranteed to be linearly independent. A single OFF test point is selected, whose projection on the given border is a convex combination of these N points. In addition, as in the two-dimensional case, the OFF point must also satisfy the inequality constraints corresponding to all adjacent borders.

The domain testing strategy requires at most $s(N + 3)$ test points per domain, where N is the dimensionality of the input space in which the domain is defined and s is the number of border segments in the boundary of the specific domain. Again, however, we can reduce this testing cost by using extreme points as ON test points and by sharing test points between adjacent domains. The following result from White and Cohen (1980) summarizes the results for domain testing: "For linearly domained programs, with each OFF point chosen a distance ϵ from the corresponding border, the domain testing strategy is guaranteed to detect all errors of magnitude greater than ϵ using no more than $s(N + 3)$ test points per domain, where N indicates the dimensionality of the input space and s is the number of predicates along the path to be tested."

6.2.5 *Some Observations on Domain Testing*

One of the major results of domain testing is that, subject to the assumption of a linearly domained program, reliable detection of domain errors requires a reasonable number of test points for a single path. This number of test points grows only linearly with the number of predicates along the path and the number of input variables. Specifically, for linearly domained programs, all domain errors can be detected using no more than $s(N + 3)$ test points per domain. However, the total cost is unacceptable for any practical program, as it will routinely contain an excessive number of paths, due to the presence of iteration loops.

A specific method by which to select the $N \times 1$ test paths for each border segment is given in Perera and White (1985). Clarke *et al.* (1982) have indicated that this $N \times 1$ test-selection strategy may result in inordinately large domain errors that remain undetected. They have suggested two improved selection techniques. The $N \times N$ selection strategy selects N ON points and N OFF points, each OFF point located at ϵ distance from the border segment. The $V \times V$ strategy selects an ON point at each vertex (or close to each vertex) on the border segment, with V OFF points, each located on a hyperplane ϵ from the border segment.

One way to view the results from domain testing is to observe that the number of test points required is a minimum for reliable detection of domain errors, and if coincidental correctness should occur, then even more test points would be required. In many places in the research testing literature, however, one finds reference to choosing only one test data point per path when a path-oriented strategy is utilized. This work shows clearly that in general this is inadequate for even a modest attempt at reliable testing.

Although we know that the problem of reliable test data generation is unsolvable, the domain testing research has shown that if attention is focused upon specific types of errors and a characterized subset of programs, reliable testing conditions can be obtained. Indeed, the problem here was to find the minimum set of conditions so that domain errors could be reliably detected.

Domain testing is an example of a structural approach, which uses only information from the program to be tested. Thus it is clear why only domain errors can be reliably detected, since they are intimately related to the structure of the given program. In order to detect computation errors or missing-path errors, we must obtain additional information, e.g., from the program specifications. This is precisely the approach of the research described in Section 6.4.

6.3 The Sufficient Paths Criterion

Although the number of required test points for each path in the domain strategy grows only linearly with the number of input variables and predicates along the path, the problem with this approach is that the number of paths grows in a highly combinatorial fashion and is potentially infinite. Moreover, many path-oriented strategies suffer from this basic problem.

In the definition of any automated path selection strategy, the questions that naturally arise are, "When does testing stop?" and "At what point is it possible to point to a particular program construct and say that it has been sufficiently tested, i.e., no errors remain undetected?" In general, we know that this problem has been proved undecidable, but a programmer's intuition suggests that such claims should be possible after the selection of a small number of test paths, especially if we possess a strategy in which we have specific confidence in terms of its ability to detect certain types of errors in some construct along that path for an appropriately restricted class of programs.

Zeil has developed a vector-space model for predicate errors (Zeil and White, 1981), for assignment errors (Zeil, 1983b), and for computation errors (Zeil, 1984), and this model has indicated substantive answers to these questions. It should be emphasized that this research and these results are essentially independent of the domain strategy and require only a path-oriented testing strategy that will reliably detect either domain errors or computation errors. Much of the material in this section has been drawn from White *et al.* (1981) and reprinted by permission of North–Holland Publishing Co., copyright © 1981.

6.3.1 *Sufficient Testing Sets*

In order to state these results more precisely, let us carefully define these questions and concepts. A set of paths is a *sufficient set* for a program construct if the failure to detect some error in that construct, using a reliable method of selecting data points along those paths, implies that this error would go undetected for any path through the program. We can then restate the questions more rigorously, as:

a) After a number of paths that pass through the construct have been tested, what is the marginal advantage of testing another path?
b) Is there a point (before nearly all paths have been tested) at which we may say that no more paths through some program construct need be chosen and tested, i.e., that this construct has been sufficiently tested?

6.3.2 *Types of Testing Blindness*

In order for us to characterize the minimal number of paths which must be tested, we first must clearly understand why multiple paths might be needed to detect an error in a construct (such as a predicate). The examples in Table I show three different reasons why a single path may not detect an erroneous predicate. These are termed *assignment blindness*, *equality blindness*, and *self-blindness*, and represent a seemingly pathological set of values for variables along the path, so that both the correct and the incorrect predicates evaluate to equal values.

TABLE I

A. ASSIGNMENT BLINDNESS

Correct	Incorrect
$A = 1$ ⋮ IF $B > 0$ THEN ⋮	$A = 1$ ⋮ IF $B + A > 1$ THEN ⋮

B. EQUALITY BLINDNESS

Correct	Incorrect

IF $D = 2$ THEN
⋮
IF $C + D > 3$ THEN
⋮
	IF $D = 2$ THEN
	⋮
	IF $C > 1$ THEN
	⋮

C. SELF-BLINDNESS

Correct	Incorrect
$X = A$ ⋮ IF $X - 1 > 0$ ⋮	$X = A$ ⋮ IF $X + A - 2 > 0$ ⋮

6.3.3 *Results from the Vector Space Model*

The vector space model has yielded an insight as to how multiple paths through a single construct can resolve the ambiguities due to various types of blindness. Results have been obtained for the effects of assignment errors in linearly domained programs and for the effects of predicate errors in more general vector spaces, for which Zeil (1983b) has characterized "vector-bounded programs." This has allowed the generalization of linear functions to polynomial or multinomial functions, for example. In his work on computation errors, Zeil (1984) also generalized his results to programs in which the computation assignment statements possessed errors that could be modeled as vector spaces.

To get a more specific and simpler intuition for his results, assume a linearly domained program; the vector space in question is then composed of

- one vector for each assigned program variable, for a total of M;
- one vector for each equality restriction on the path domain, at most N total, where N is the number of input variables.

The results of this research, which provides answers to questions (a) and (b) posed in Section 6.3.1 can be stated as follows for predicate errors and assignment errors:

> For any predicate in a linearly domained program, the smallest sufficient set of test paths will contain at most $(M + N + 1)$ paths; if a set of paths has been tested, which paths pass through the predicate of interest, a simple vector criterion, in Zeil and White (1981), will determine whether a proposed additional path is required to detect an error in that predicate.
>
> A minimal set of paths sufficient for testing a given sequential set of assignment statements in a linearly domained program will contain at most $M(M + N + 1)$ paths; if a set of paths has been tested, which paths pass through those assignment statements, a simple vector criterion, in Zeil (1981), will determine whether a proposed additional path is required to detect an error in those assignment statements.

Notice the substantially larger number of paths to detect assignment errors as opposed to predicate errors; both path measures are polynomial in M and N. Linearly domained programs have been assumed to achieve this result. Another serious problem, however, is that those vector criteria can only be applied post hoc to a selected path in conjunction with a set of previously selected paths. In the next section we will describe some experiments aimed at a solution to this problem.

6.3.4 *Experiments Using the Sufficient-Paths Criterion*

Experiments were conducted on linearly domained programs, by testing predicates using Zeil's sufficient-paths vector space criterion; these experiments are reported in White and Sahay (1985). Nine programs were used; they were linearly domained and tended either to be data-structuring programs (to perform sorting, searching, or set operations) or number-theoretic programs (such as Euclid's greatest common divisor or greatest common factor).

Zeil and White (1981) showed that each predicate to be tested is associated with an error space of maximum dimension $(M + N + 1)$, where N is the number of input variables and M the number of program variables. As each subsequent path through that predicate is chosen for testing, the resultant error space is reduced in dimension (or else that path is discarded and not tested). One of the primary reasons for these experiments was to develop heuristics for path selection to effect the most rapid reduction of predicate error spaces. Another reason for these experiments was to characterize the *irreducible error space* of a predicate, which cannot be reduced further by the selection of any other path through that predicate.

Zeil's sufficient-paths criterion indicates that an upper bound of $(M + N)$ on the number of paths required for testing a predicate and $p(M + N)$ paths would be required to test all p predicates in a program if each predicate were tested independently. Another objective of these experiments was to see how many paths were required for each of these cases.

Several heuristics emerged from these experiments, and they lead to improved path selection. In White and Sahay (1985), it is shown that the minimum sufficient set of paths required for testing a given predicate in a linearly domained program will contain at most $(M + Z + 1)$ paths, where Z is the number of independent equality restrictions encountered along the first path chosen for that predicate. Thus the first path should be selected through the given predicate with the fewest possible equality restrictions. In the experiments this was done, so $Z = 0$, and then $(M + 1)$ paths were typically chosen, showing this to be a tight bound. Subsequent paths for that predicate should be selected so as to eliminate vectors in the resulting error space.

In selecting predicates to test, those with the fewest paths leading to them should be chosen first; these tend to be predicates near the beginning of the program. Paths should be considered that are extensions of subpaths previously chosen for earlier predicates; these experiments have shown that this approach was quite successful, leading to a number of paths for testing all predicates which was at most only one more than the bound $(M + Z + 1)$ for a single predicate, far below the predicted bound of $p(M + N)$. It remains for further experimentation to see if these results will hold in general.

The irreducible error spaces in the experiments were found to contain unused variables, equality restrictions, and invariant expressions. One should be able to predict, a priori, the occurrence of the first two items; invariant expressions will require further study.

6.3.5 *Loop Iteration Limits*

Another result of Zeil's work on sufficient testing is to address the problem of iteration loop limits. Many testing researchers and practitioners have recommended that no more than k iterations of any loop be used for testing, but without any logical or theoretical justification. Zeil (1981) shows that if there is a path S with more than $(M + N)$ iterations of some single-entry, single-exit loop and a predicate error e detectable along S, then there exists another path S' with no more than $(M + N)$ iterations of that loop, such that, ignoring coincidental equalities, e is detectable along S'; N represents the number of input variables and M the number of program variables.

Zeil also makes a number of observations about this result, called the "Iteration Limit Theorem," which we should indicate here. This theorem guarantees only that a short path S' exists for which the error is detectable, but does not guarantee that the path S' is feasible or that the predicate interpretation involved is nonredundant. He also observes that, since the proof was independent of the surrounding program structure, the result holds regardless of the existence of nested iteration loops. The Iteration Limit Theorem applies, then, for the total number of iterations of an innermost loop in this nesting structure.

Another result in a similar vein and applying to domain testing has been presented by Wiszniewski (1985) and is called "iterated domains." In this paper he defines *elementary classes of paths*; an *elementary path* is a control path that does not execute any instruction more than once (i.e., it does not contain any iteration loops). Each elementary class of paths will then be composed of the elementary path and all paths that differ from that path only by the addition of an arbitrary number of loop iterations. This is the same notion as given by Howden (1981) some time ago and called "boundary-interior classes of paths."

Wiszniewski then goes on to prove that if the computations in a program P are primitive recursive, then only a finite number of paths from each elementary class need be tested. He argues that practical programs compute primitive recursive functions when the number of iterations of loops can be bounded.

Further research is needed to see whether this approach can be applied in practice. First one must be able to identify which paths to select to be tested. Next it must be mentioned that the number of elementary classes can grow exponentially even in small programs, an observation also made by Howden (1981). This result could certainly prove very useful in domain testing or in other testing methods where the number of control paths grows in an unacceptable combinatorial manner.

6.3.6 Perturbation Testing

Based upon the results of sufficient paths, Zeil has developed a new approach to testing called "perturbation testing." This has been applied to program assignment statements (Zeil, 1983b), computation errors (Zeil, 1984), and domain errors (Zeil, 1983a). In his previous results on sufficient paths, Zeil based the theory upon knowledge about the programs being tested (e.g., linearly domained, vector bounded, etc.). In practice we usually do not have this information. Perturbation testing only requires that the user make specific assumptions about the functional form of the error terms as perturbations. Thus one could begin by testing to eliminate all linear error terms and later expand this testing to cover higher-order error terms.

As an example, consider perturbation testing for computation errors (Zeil, 1984); Zeil compares this approach to *algebraic testing* as described by Howden (1978a). Zeil indicates that algebraic testing can be considered as a black box approach, where an a priori prescription is given to detect a computation error. For example, the class of multinomial functions on r variables with exponents less than q can be tested using q^r points arranged in a configuration called a "cascade set" (Howden, 1978a). With perturbation testing applied to the same multinomial functions, the output is observed with each test, a blindness error space computed, and test data subsequently selected that will substantially reduce that error space.

Zeil argues that algebraic testing is mathematically equivalent to this approach, but that perturbation testing offers more flexibility. He gives a multinomial example with three variables and maximum total exponent of three in one term; although this requires twenty-seven test points with algebraic testing, Zeil only needs nine test points. Of course, there is the additional computational work to construct and examine the blindness error space.

Until perturbation testing receives more use in practice, it remains to be seen whether practitioners can benefit from the insights provided by this highly structured testing approach.

6.4 Program Specification Testing and Partition Analysis

We have indicated that one of the primary limitations of structured testing methods, which include path-oriented techniques, is that they use only the program itself. A number of researchers are actively examining the possibility of generating test data from program specifications, especially to complement structural approaches such as path testing.

Gourlay (1983) has recently surveyed many aspects of the program testing problem and has included an area he calls "specification-dependent testing." He has pointed out that, due to the lack of one pervasive specification language, each research investigator utilizes his or her own specification language and thus various methods cannot be fairly compared. These specification languages range from very formal systems such as the predicate calculus to more procedural specification languages, which have been used to generate test data in addition to the computer programs themselves.

Cartwright (1981) has developed a very high-level language with which to express program specifications, and since it is procedural, this language allows him to generate test data from the specifications. Richardson and Clarke (1985) have also chosen to use a very high-level language for program specifications, and explicitly perform a path analysis of the specification to obtain a partition of the input space, which is used to further refine the path-testing partition from the original program. Gourlay (1981) has shown that specifications can be written using the flexibility and power of the predicate calculus, and yet test data can also be generated from specifications expressed in this more formal structure.

There are a number of executable functional specifications described in the literature. One example is provided by a system called DESCARTES, reported by Urban (1982). The specifications are developed using this system in a top-down modular approach. Partial specifications can be executed on the language processor through abstract execution. This has significant software development advantages, since the developer, with user approval, can proceed with software development convinced that the specification corresponds to what is required.

At the University of Maryland, Gannon, McMullin, and Hamlet (1981) have developed a compiler-based system, DAISTS (Data-Abstraction Implementation, Specification, and Testing System), that combines a data-abstraction implementation language with specification by algebraic axioms. This system was based on earlier conceptual work by Hamlet (1977a). In this system both verification and testing approaches are used. The verification is used to see if the axioms and implementation agree. Structural testing is applied to both code and axioms to evaluate the test data, and the axioms serve as the test oracle. The user writes specification axioms, the implementation, and test data; the system furnishes the test driver and evaluation of correctness. The authors indicate that the test data provides a severe exercise for the axioms or implementation or both.

This DAISTS system was applied to a practical example involving a record-oriented text editor, as documented in a paper by McMullin and Gannon (1983). The specification for this editor was written using algebraic axioms and serves as an oracle for judging the correctness of values returned by the functions of the implementation on the user inputs. This system should be even more impressive as it is applied to other implementation examples.

Each of these research efforts makes a contribution to specification testing. The system of Richardson and Clarke, together with the DAISTS and DESCARTES systems, provides the distinct advantage of a working system with which these researchers can conduct experiments to evaluate various approaches to specification and program testing. Richardson and Clarke (1981) best illustrate how to integrate both verification and testing, so we will focus on this integrated system for the remainder of this section, as an example of a specification testing technique. We will see the explicit use of many concepts and ideas previously studied.

6.4.1 *Overview of Partition Analysis*

The partition analysis system, first proposed by Richardson and Clarke (1981) and then later reported after extensive development and experimentation (Richardson and Clarke, 1985), obtains information from both the specification and the implementation. Much of the material in Section 6.4 is drawn from Richardson and Clarke (1985), and reprinted with permission of IEEE, copyright © 1985. Their method is applicable to various specification languages, both procedural and nonprocedural.

Partition analysis can be described in three steps:

1. Symbolic evaluation and other analysis techniques are used to determine the partition, obtaining the input points in each subdomain for both specification and implementation. The computation in each of these subdomains is obtained for both the specification and implementation.
2. Symbolic evaluation and other verification techniques are applied to the two computational descriptions, to determine equality over the specified subdomains. This will require additional redefinition of the subdomains, to reconcile any detected differences in the subdomains.
3. The subdomain and computational descriptions are used to derive test data. Additional test data may be developed for those subdomains where verification was not successful.

The first objective of partition analysis is to divide or partition the procedure domain into more manageable subdomains. The second objective is to partition the set of input data into subdomains so that the elements of each subdomain are accounted for uniformly by the specification and processed uniformly by the implementation.

The partition analysis system can handle both high-level formal specification languages based upon predicate calculus or state transformation methods, and low-level procedural languages. Richardson and Clarke (1985) have developed their own high-level language, called SPA, which is an extended PDL/ADA language (Kerner, 1983). The partition analysis approach seems not to work well with algebraic or axiomatic specifications.

6.4.2 *Symbolic Evaluation and Verification*

The authors use symbolic evaluation as a means to verify both that subdomains defined by the specification and the implementation are the same, and that computations for a subdomain are the same from those two sources. If the subdomains are not the same, then further partitioning may be necessary. If computations can be proved different, then the implementation differs from the specification and one or the other is in error. Richardson and Clarke possess considerable expertise with symbolic evaluation techniques; a symbolic execution system was developed by Clarke (1976) to generate test data, and a survey paper on the subject of symbolic evaluation was also provided (Clarke and Richardson, 1981).

These authors also use iteration loop analysis and symbolically represent iteration loops through a closed-form expression that captures the effect of that loop. This requires the derivation and solution of recurrence relations, which represent the changes to program variables made by iterations of the loop. For a discussion of these techniques see Cheatham *et al.* (1979) or Clarke and Richardson (1981).

It is not always possible to solve these recurrence relations or to prove that two computations or subdomain descriptions are the same. (It should be recognized that problems in the latter set are in general undecidable, from our discussion in Section 2.2.) At that point the verification has failed for these subdomains, and the issue is turned over to the testing phase of the system. Clarke and Richardson (1985) have observed that this occurs in surprisingly few cases for practical software systems.

6.4.3 *Partition Testing*

One could ask why partition testing would be necessary if the verification process were successful. We have already noted that verification is not always successful for all subdomains. In addition, a run-time environment is utilized for testing, rather than a conceptual environment as was used for verification. Another problem, however, is that it is unreasonable to assume that specifications are always correct or complete, and testing can bring this problem into focus.

Partition testing uses a combination of structural and functional methods to detect both computation errors and domain errors. Computation errors are detected by functional testing (Howden, 1980), including special value testing and extremal output value testing, as well as by a method proposed by Redwine (1983). Domain errors are detected by domain testing (White and Cohen, 1980), boundary value and condition coverage (Myers, 1979), and extremal input value testing (Howden, 1980). An objective is to unify these techniques so that these criteria can be automated. Notice that missing-path errors can also be detected, since specification information is provided.

6.4.4 *Evaluation of Partition Analysis*

Richardson and Clarke (1985) reports an extensive experimentation with the partition analysis system using thirty-four diverse programs. Documented errors in these programs were systematically detected by either symbolic evaluation or testing or both, including four missing-path errors. There were a number of instances where verification failed; in each of those cases the testing was able to discover any existing errors. Since most of these programs were either correct or had few errors, some method was sought to demonstrate the

effectiveness of the test set. It is interesting that mutation analysis was applied, and for all thirty-four programs, the mutant programs were either killed by testing or were proved to be equivalent to the original program by the verification technique using symbolic evaluation.

7. Conclusions and Future Prospects

We have seen that, although many essential problems in program testing are undecidable, there has been notable progress with a number of testing approaches. This has been achieved by concentrating on certain classes of programs and also on the detection of certain types of errors. If we recall the approach to symbolic testing by Richardson and Clarke (1985), even though they were faced with potentially undecidable issues, they have succeeded through persistence and occasionally accepting less than total success in comparing two symbolic formulas. Their system is designed so as to not only recover from this failure, but to use it to positive advantage in a later testing phase.

We have identified as another question of major economic importance the need for a testing oracle to determine correctness. In practice this is one of the most difficult and costly problems, and it is not just a theoretical issue. Some progress has been made with executable specifications, but considerably more advancement is needed before we can claim to have automated this difficult aspect of testing.

There are many controversial issues in testing where both practitioners and researchers would not agree, but one area of common agreement by all knowledgeable software experts is that static testing should always be done prior to dynamic testing, for the advantages gained in the early detection of errors far outweighs its cost, which is low. Yet very few software projects perform systematic static testing.

Functional testing has been applied in practice without useful guidelines or a solid theoretical basis. This has to a large extent been provided by Howden in his work on functional testing (1980; 1985) and also by his research on weak mutation testing (Howden, 1982). These guidelines and research results should be implemented in software project test plans.

7.1 Future Prospects

Practical software projects have adopted the concepts of statement and branch coverage, best discussed by Myers (1979), as simple test plans for structural testing. There is considerable research activity in the area of data-flow testing, in which these simpler coverage measures are generalized to the more powerful data-flow coverage criteria. The most recent work has been reported by Frankl and Weyuker (1985) and Clarke *et al.* (1985). More experimental work is needed with these systems, and this improved testing technology needs to move out to practitioners and into current software project test plans.

We have seen the limitations of structural testing, yet this is one of the powerful testing tools available to the practitioner. The partition analysis system of Richardson and Clarke (1985) is one of the best examples of the unification of many techniques and approaches into a testing and verification system; they even utilized mutation testing to better evaluate the test set. There are still interesting research issues in structural testing, in the selection of the best paths, where the work on sufficient paths (Zeil, 1981) and on perturbation testing (Zeil, 1983b; 1984) needs to be extended to select best paths for any type of structural testing (including data-flow testing).

Another area of substantial research activity we have not mentioned is the application of logic programming to the problem of test-data generation. At one level this involves developing test-case specifications and implementing them in PROLOG (e.g., Ural and Probert, 1984). At a higher level, with more interesting consequences, logic programming can make a contribution to testing by deriving functional test data sets from a formal specification in PROLOG and a tool based on logic programming. A number of researchers are working in this area, including Gerhart (1985), Bouge (1985), Bouge *et al.* (1985), and on the industrial side, Pesch *et al.* (1985).

There is continuing research generally on formal specifications, and from the perspective of testing, the work on executable specifications is most needed. In Section 6.4 we identified the work of Gannon *et al.* (1981) as quite promising in this regard. Recently Day and Gannon (1985) reported on a test oracle based on formal specifications that was used in an introductory computer science course. Although the students in such a course are unfamiliar with testing methodology, they were required to state the specifications of their computer program assignments using the specification system developed, which utilized a BNF grammar. The authors report that most of the students were able to accomplish this task successfully.

This illustrates what we will expect of testing and verification in the next decade: Methodology now in the research phase will become embedded in software technology and will become pervasive, providing testing and verification tools that we can take for granted in terms of their efficiency and effectiveness.

References

Beizer, B. (1983). "Software Testing Techniques." Van Nostrand Reinhold, New York.

Bouge, L. (1985). A contribution to the theory of program testing. *Theor. Comput. Sci.* 37 (2), 151–181.

Bouge, L., Choquet, N., Fribourg, L., and Gaudel, M. C. (1985). Application of PROLOG to test sets generation from algebraic specifications. *In* "Formal Methods and Software Development," Vol. 2 (H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, eds.), pp. 261–274. Springer Verlag, Berlin.

Boyer, R. S., Elspas, B., and Levitt, K. N. (1975). SELECT—A formal system for testing and debugging programs by symbolic execution. *Proc. Int. Conf. Reliable Software*, pp. 234–245.

Brainerd, W. S., and Landweber, L. H. (1974). "Theory of Computation." Wiley, New York.

Budd, T. A. (1981). Mutation analysis: Ideas, examples, problems and prospects. *In* "Computer Program Testing" (B. Chandrasekran and S. Radicchi, eds.), pp. 129–148. North–Holland, Amsterdam.

Budd, T. A., DeMillo, R., Lipton, R. J., and Sayward, F. G. (1978). The design of a prototype mutation system for program testing. *Proc. ACM Nat. Comput. Conf.*, pp. 623–627.

Cartwright, R. (1981). Formal program testing. *Proc. 8th Annu. ACM Symp. Principles Program. Lang.*, pp. 126–132.

Cheatham, T. E., Holloway, G. H., and Townley, J. A. (1979). Symbolic evaluation and the analysis of programs. *IEEE Trans. Software Eng.* **SE-5** (4), 402–417.

Clarke, L. A. (1976). A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.* **SE-2** (3), 215–222.

Clarke, L. A., and Richardson, D. J. (1981). Symbolic evaluation methods—Implementations and applications. *In* "Computer Program Testing" (B. Chandrasekaran and S. Radicchi, eds.), pp. 65–102. North–Holland, Amsterdam.

Clarke, L. A., Hassell, J., and Richardson, D. J. (1982). A close look at domain testing. *IEEE Trans. Software Eng.* **SE-8** (4), 380–390.

Clarke, L. A., Podgurski, A., Richardson, D. J., and Zeil, S. J. (1985). A comparison of data flow path selection criteria. *Proc. 8th Int. Conf. Software Eng.*, pp. 244–251.

Davis, M. (1973). Hilbert's tenth problem is unsolvable. *Amer. Math. Mon.* **80**, 233–269.

Day, J. D., and Gannon, J. D. (1985). A test oracle based on formal specifications. *Proc. SoftFair II*, pp. 126–130.

DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer* **11** (4), 34–41.

Duran, J. W., and Ntafos, S. C. (1981). A report on random testing. *Proc. 5th Int. Conf. Software Eng.*, pp. 179–183.

Fosdick, L. D., and Osterweil, L. J. (1976). Data-flow analysis in software reliability. *ACM Comput. Surv.* **8** (3), 305–330.

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

Frankl, P. G., and Weyuker, E. J. (1985). A data-flow testing tool. *Proc. SoftFair II*, pp. 46–53.

Gannon, J., McMullin, P., and Hamlet, R. (1981). Data-abstraction implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.* **3** (3), 211–223.

Gerhart, S. (1985). Test data generation using PROLOG. Tech. Report No. 2, Wang Institute of Graduate Studies, Tyngsboro, Mass.

Goodenough, J. B., and Gerhart, S. L. (1975). Toward a theory of test data selection. *IEEE Trans. Software Eng.* **SE-1** (2), 156–173.

Gourlay, J. S. (1981). Theory of testing computer programs. Ph.D. dissertation, Department of Computer and Communication Sciences, The University of Michigan, Ann Arbor, Michigan.

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

Hamlet, R. (1977a). Testing programs with finite sets of data. *Comput. J.* **20** (3), 232–237.

Hamlet, R. (1977b). Testing programs with the aid of a compiler. *IEEE Trans. Software Eng.* **SE-3** (4), 279–290.

Hecht, H. S. (1977). "Flow Analysis of Computer Programs." North–Holland, Amsterdam.

Howden, W. E. (1976). Reliability of the path analysis testing strategy. *IEEE Trans. Software Eng.* SE-2 (3), 208–215.

Howden, W. E. (1978a). Algebraic program testing. *ACTA Inform* **10**, 53–66.

Howden, W. E. (1978b). A survey of dynamic analysis methods. *In* "Tutorial: Software Testing and Validation Techniques" (E. Miller and W. E. Howden, eds.), pp. 209–231. IEEE Computer Society, Washington, D. C.

Howden, W. E. (1980). Functional program testing. *IEEE Trans. Software Eng.* **SE-6** (2), 162–169.

Howden, W. E. (1981). Completeness criteria for testing elementary program functions. *Proc. 5th Int. Conf. Software Eng.*, pp. 235–243.

Howden, W. E. (1982). Weak mutation testing and completeness of program test sets. *IEEE Trans. Software Eng.* **SE-8** (4), 371–379.

Howden, W. E. (1985). The theory and practice of functional testing. *IEEE Software* **2** (5), 6–17.

Huang, J. C. (1975). An approach to program testing. *Comput. Surv.* **7**, 113–128.

Huang, J. C. (1979). Detection of data flow anomaly through program instrumentation. *IEEE Trans. Software Eng.* SE-5 (3), 226–236.

"IMSL Library Reference Manual," Int. Math. and Statist. Libraries, Houston. 1978.

Kerner, J. S. (1983). Design methodology subcommittee chairperson's letter and matrix. *Ada Lett.* **2** (6), 110–115.

Korel, B., and Laski, J. (1985). A tool for data flow oriented program testing. *Proc. SoftFair II*, pp. 34–37.

Laski, J. W., and Korel, B. (1983). A data flow oriented program testing strategy. *IEEE Trans. Software Eng.* SE-9 (3), 347–354.

McMullin, P. R., and Gannon, J. D. (1983). Combining testing with formal specifications: A case study. *IEEE Trans. Software Eng.* SE-9 (3), 328–334.

Miller, E. F., and Howden, W. E. (1981). "Tutorial: Software Testing and Validation Techniques," second ed. IEEE Computer Society Press, Los Alamitos, CA.

Myers, G. J. (1979). "The Art of Software Testing." Wiley, New York.

Ntafos, S. (1984). On required element testing. *IEEE Trans. Software Eng.* **SE-10** (6), 795–803.

Ntafos, S. C., and Hakimi, S. L. (1979). On path cover problems in digraphs and applications to program testing. *IEEE Trans. Software Eng.* SE-5 (5), 520–529.

Osterweil, L. J., and Fosdick, L. D. (1978). DAVE—A validation, error detection and documentation system for FORTRAN programs. *Software Pract. Experience.* **6**, 473–486.

Osterweil, L. J., Fosdick, L. D., and Taylor, R. N. (1981). Error and anomaly diagnosis through data flow analysis. *In* "Computer Program Testing" (B. Chandrasekaran and S. Radicchi, eds.), pp. 35–63. North–Holland, Amsterdam.

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

Parnas, D. L. (1972). A technique for software module specification with examples. *Commun. ACM* **15** (5), 330–336.

Perera, I. A., and White, L. J. (1985). Selecting test data for the domain testing strategy. Tech. Report TR-85-5, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada.

Pesch, H., Schnupp, P., Schaller, H., and Spirk, A. P. (1985). Test case generation using PROLOG. *Proc. 8th Int. Conf. Software Eng.*, pp. 252–258.

Ramamoorthy, C. V., and Kim, K. H. (1975). Optimal placement of software monitors aiding systematic testing. *IEEE Trans. Software Eng.* **SE-1** (4), 403–410.

Rapps, S., and Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE Trans. Software Eng.* **SE-11** (4), 367–375.

Redwine, S. T. (1983). An engineering approach to test data design. *IEEE Trans. Software Eng.* SE-9 (2), 191–200.

Richardson, D. J., and Clarke, L. A. (1981). A partition analysis method to increase program reliability. *Proc. 5th Int. Conf. Software Eng.*, pp. 244–253.

Richardson, D. J., and Clarke, L. A. (1985). Partition analysis: A method of combining testing and verification. *IEEE Trans. Software Eng.* **SE-11** (12), 1477–1490.

Stucki, L. G. (1977). New directions in automated tools for improving software quality. *In* "Current Trends in Programming Methodology, Vol. 2: Program Validation" (R. T. Yeh, ed.), pp. 80–111. Prentice Hall, Englewood Cliffs, New Jersey.

Taylor, R. N., and Osterweil, L. J. (1980). Anomaly detection in concurrent software by static data flow analysis. *IEEE Trans. Software Eng.* **SE-6** (3), 265–278.

Ural, H., and Probert, R. (1983). User-guided test sequence generation. *In* "Protocol Specification, Testing and Verification III" (C. Sunshine, ed.), pp. 421–436. North–Holland, Amsterdam.

Urban, J. E. (1982). Software development with executable function specifications. *Proc. 6th Int. Conf. Software Eng.*, pp. 418–419.

Weyuker, E. J. (1979). The applicability of program schema results to programs. *Int. J. Comput. Inf. Sci.* **8**, 387–403.

Weyuker, E. J. (1984). The complexity of data flow criteria for test data selection. *Inf. Process. Lett.* **19** (2), 103–109.

Weyuker, E. J., and Ostrand, T. J. (1980). Theories of program testing and the application of revealing subdomains. *IEEE Trans. Software Eng.* SE-6 (3), 236–246.

White, L. J. (1981). Basic mathematical definitions and results in testing. *In* "Computer Program Testing" (B. Chandrasekaran and S. Radicci, eds.), pp. 13–24. North–Holland, Amsterdam.

White, L. J., and Cohen, E. I. (1980). A domain strategy for computer program testing. *IEEE Trans. Software Eng.* **SE-6** (3), 247–257.

White, L. J., and Sahay, P. N. (1985). Experiments determining best paths for testing computer program predicates. *Proc. 8th Int. Conf. Software Eng.*, pp. 238–243.

White, L. J., Teng, F. C., Kuo, H., and Coleman, D. (1978). An error analysis of the domain testing strategy. Tech. Report CISRC-TR-78-2, Ohio State University, Columbus, Ohio.

White, L. J., Cohen, E. I., and Zeil, S. J. (1981). A domain strategy for computer program testing. *In* "Computer Program Testing" (B. Chandrasekaran and S. Radicchi, eds.), pp. 103–113. North–Holland, Amsterdam.

Wiszniewski, B. W. (1985). Can domain testing overcome loop analysis? *Proc. 9th Comput. Software Appl. Conf.*, pp. 304–309.

Woodward, M. R., Hedley, D., and Hennell, M. A., (1980). Experience with path analysis and testing of programs. *IEEE Trans. Software Eng.* **SE-6** (3), 278–286.

Zeil, S. J. (1981). Selecting sufficient sets of test paths for program testing. Ph.D. dissertation, Ohio State University, Columbus, Ohio. Tech. Report OSU-CISRC-TR-81-10.

Zeil, S. J. (1983a). Perturbation testing for domain errors. Tech. Report 83-38, Univ. Mass., Amherst, Mass.

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

Zeil, S. J. (1984). Perturbation testing for computational errors. *Proc. 7th Int. Conf. Software Eng.*, pp. 257–265. Tech. Report 83-23 (1983), Univ. Mass., Amherst, Mass.

Zeil, S. J., and White, L. J. (1981). Sufficient test sets for path analysis testing strategies. *Proc. 5th IEEE Int. Conf. Software Eng.*, pp. 184–191.