Validation, Verification, and Testing of Computer Software

W. RICHARDS ADRION

Division of Mathematical and Computer Sciences, National Science Foundation, Washington, D.C. 20550

MARTHA A. BRANSTAD

Institute for Computer Science and Technology, National Bureau of Standards, Washington, D.C. 20234

AND

JOHN C. CHERNIAVSKY

Division of Mathematical and Computer Sciences, National Science Foundation, Washington, D.C. 20550

Software quality is achieved through the application of development techniques and the use of verification procedures throughout the development process Careful consideration of specific quality attributes and validation requirements leads to the selection of a balanced collection of review, analysis, and testing techniques for use throughout the life cycle. This paper surveys current verification, validation, and testing approaches and discusses their strengths, weaknesses, and life-cycle usage. In conjunction with these, the paper describes automated tools used to implement validation, verification, and testing. In the discussion of new research thrusts, emphasis is given to the continued need to develop a stronger theoretical basis for testing and the need to employ combinations of tools and techniques that may vary over each application.

Categories and Subject Descriptors: D 2 1 **[Software Engineering]**: Requirements/ Specifications—*methodologies, tools;* D 2 2 **[Software Engineering]**: Tools and Techniques—*decision tables; modules and interfaces, structured programming; top-down programming; user interfaces;* D.2.3 **[Software Engineering]**: Coding—*standards;* D.2.4 **[Software Engineering]**. Program Verification—*assertion checkers, correctness proofs; reliability, validation;* D.2.5 **[Software Engineering]** Testing and Debugging—*debugging aids; monitors; symbolic execution; test data generators;* D.2.6 **[Software Engineering]** Programming Environments, D.2.7 **[Software Engineering]**: Distribution and Maintenance—*documentation; version control;* D.2.8 **[Software Engineering]**: Metrics—*complexity measures;* D.2.9 **[Software Engineering]**: Management—*life cycle; programming teams; software configuration management, software quality assurance (SQA)*

General Terms: Reliability, Verification

INTRODUCTION

Programming is an exercise in problem solving. As with any problem-solving activity, determination of the validity of the solution is part of the process. This survey discusses testing and analysis techniques that can be used to validate software and to instill confidence in the quality of the programming product. It presents a collection of verification techniques that can be used throughout the development process to facilitate software quality assurance.

Programs whose malfunction would have severe consequences justify greater effort in their validation. For example, software used in the control of airplane landings or directing of substantial money transfers requires higher confidence in its proper functioning than does a car pool locator program. For each software project, the validation requirements, as well as the product requirements, should be determined and specified at the initiation of the project. Project size, uniqueness, criticalness, the

CONTENTS

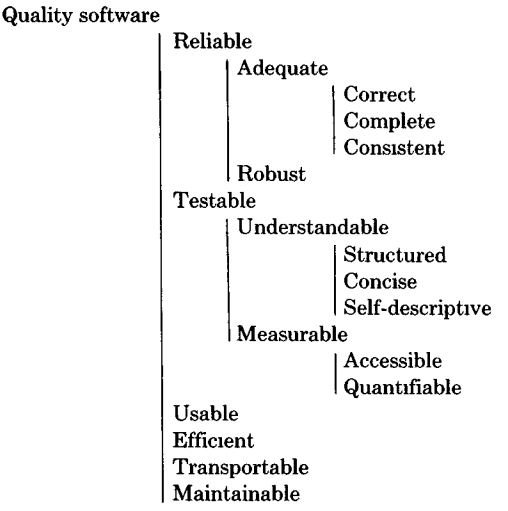

Figure 1. A hierarchy of software quality attributes

cost of malfunction, and project budget all influence the validation needs. After the validation requirements have been clearly stated, specific techniques for validation, verification, and testing (VV&T) can and should be chosen. This paper concentrates on VV&T in medium and large projects, but many of the individual techniques are also applicable to small projects. VV&T for very small projects is discussed in BRAN80.

Some of the terms used in this article may appear to have slightly different meanings elsewhere in the literature. For that reason, a glossary is included.

Verification, validation, and testing are closely tied to *software quality*. There have been many studies directed toward determining appropriate factors for software quality [BOEH78, MCCA77, JONE76]. A number of attributes have been proposed; the set given by Figure 1 is representative. Each major quality attribute is given at the left of the figure and its characterizations are placed below and to the right of it. For example, software with the quality attribute of being *testable* has the characterization of being both *understandable* and *measurable,* where understandable software has, in turn, the further characterizations of being *structured, concise,* and *self-descriptive.* Most of these factors are qualitative rather than quantitative.

The main attributes of software quality include reliability, testability, usability, efficiency, transportability, and maintainability, but in practice, efficiency often conflicts with other attributes. For example, using a vendor-specific FORTRAN feature may increase execution efficiency but decrease code transportability. Each software development project must determine which factors have priority and must specify their relative importance.

Two quality factors, reliability and testability, are tightly coupled with testing and verification issues. Clearly, reliable software must first be adequate: it must be correct, complete, and consistent at each stage of the development. Incomplete requirements will lead to an inadequate design and an incorrect implementation. The second reliability requirement, robustness, represents the ability of the software to continue to operate or survive within its environment.

Testable software must exhibit understandability and measurability. Understandability requires the product at each

stage to be represented in a structured, concise, and self-descriptive manner so that it can be compared with other stages, analyzed, and understood. Measurability requires means to exist for actually instrumenting or inserting probes, for testing, and for evaluating the product of each stage.

Although good quality may be difficult to define and measure, poor quality is glaringly apparent. For example, software that is filled with errors or does not work obviously lacks quality. Program testing, by executing the software using representative data samples and comparing the actual results with the expected results, has been the fundamental technique used to determine errors. However, testing is difficult, time consuming, and often inadequate. Consequently, increased emphasis has been placed upon ensuring quality throughout the entire development process, rather than trying to do so after the process is finished.

1. VERIFICATION THROUGH THE LIFE CYCLE

In this survey, we look at verification, validation, and testing techniques as they are applied throughout the software development life cycle. The traditional development life cycle confines testing to a stage immediately prior to operation and maintenance. All too often, testing is the only verification technique used to determine the adequacy of the software. When verification is constrained to a single technique and confined to the latter stages of development, severe consequences can result, since the later in the life cycle that an error is found, the higher is the cost of its correction [INFO79]. Consequently, if lower cost and higher quality are the goal, verification should not be isolated to a single stage in the development process but should be incorporated into each phase of development. Barry Boehm [BOEH77] has stated that one of the most prevalent and costly mistakes made in software projects today is deferring the activity of detecting and correcting software problems until late in the project. The primary reason for early investment in verification activity is to catch potentially expensive errors early before the cost of their correction escalates.

Life-cycle stage	Verification activities
Requirements	Determine verification approach Determine adequacy of requirements Generate functional test data
Design	Determine consistency of design with requirements Determine adequacy of design Generate structural and functional test data
Construction	Determine consistency with design Determine adequacy of implementation Generate structural and functional test data Apply test data
Operation and Maintenance	Reverify, commensurate with the level of redevelopment

Figure 2. Life-cycle verification activities

Figure 2 presents a life-cycle chart that includes verification activities. The success of performing verification throughout the development cycle depends upon the existence of a clearly defined and stated product at each development stage (e.g., a requirement specification at the requirements stage). The more formal and precise the statement of the development product, the more amenable it is to the analysis required to support verification. Many of the new software development methodologies encourage a visible, analyzable product in the early development stages.

1.1 The Requirements Definition Stage

The verification activities that accompany the requirements stage of software development are extremely significant. The adequacy of the requirements, that is, their correctness, completeness, and consistency, must be thoroughly analyzed, and initial test cases with the expected (correct) responses must be generated. The specific analysis techniques that can be applied depend upon the methodology used to specify the requirements. At a minimum, disci-

plined inspection and review should be used, with special care taken to determine that all pertinent aspects of the project have been stated in the requirements. Omissions are particularly pernicious and difficult to discover. Developing scenarios of expected system use, while helping to determine the test data and anticipated results, also help to establish completeness. The tests will form the core of the final test set. Generating these tests also helps guarantee that the requirements are testable. Vague or untestable requirements will leave the validity of the delivered product in doubt since it will be difficult to determine if the delivered product is the required one. The late discovery of requirements inadequancy can be very costly. A determination of the criticality of software quality attributes and the importance of validation should be made at this stage. Both product requirements and validation requirements should be established.

Some tools to aid the developer in requirements definition exist. Examples include Information System Design and Optimization System (ISDOS) with Program Statement Language (PSL) and Program Statement Analyzer (PSA) [TEIC77], Software Requirements Engineering Program (SREP) [ALFO77], Structured Analysis and Design Technique (SADT) [ROSS77], and Systematic Activity Modeling Method (SAMM) [LAMB78]. All provide a disciplined framework for expressing requirements and thus aid in the checking of consistency and completeness. Although these tools provide only rudimentary verification procedures, requirement verification is greatly needed and it is a central subject of research being performed by Teichroew and his group at Michigan.

Ideally, organization of the verification effort and test management activities should be initiated during the requirements stage, to be completed during preliminary design. The general testing strategy, including selection of test methods and test evaluation criteria, should be formulated, and a test plan produced. If the project size and criticality warrants, an independent test team should be organized. In addition, a test schedule with observable milestones should be constructed.

At this same time, the framework for quality assurance and test documentation should be estimated [FIPS76, BUCK79, IEEE79]. FIPS Publication 38, the National Bureau of Standards guideline for software documentation during the development phase, recommends that test documentation be prepared for all multipurpose or multiuser projects, and for all software development projects costing over $5000. FIPS Publication 38 recommends the preparation of a test plan and a test analysis report. The test plan should identify test milestones and provide the testing schedule and requirements. In addition, it should include both the specifications, descriptions, and procedures for all tests, and the test data reduction and evaluation criteria. The test analysis report should summarize and document the test results and findings. The analysis summary should present the software capabilities, deficiencies, and recommendations. As with all types of documentation, the extent, formality, and level of detail of the test documentation are dependent upon the management practice of the development organization and will vary depending upon the size, complexity, and risk of the project.

1.2 The Design Stage

During detailed design, validation support tools should be acquired or developed and the test procedures themselves should be produced. Test data to exercise the functions introduced during the design process as well as test cases based upon the structure of the system should be generated. Thus, as the software development proceeds, a more effective set of test cases is built up.

In addition to the generation of test cases to be used during construction, the design itself should be analyzed and examined for errors. Simulation can be used to verify properties of the system structures and subsystem interaction. Design walk-throughs, a form of manual simulation, can and should be used by the developers to verify the flow and logical structure of the system. Design inspection should be performed by the test team to discover missing cases, faulty logic, module interface mismatches,

data structure inconsistencies, erroneous I/O assumptions, and user interface inadequacies. Analysis techniques are used to show that the detailed design is internally consistent, complete, and consistent with the preliminary design and requirements.

Although much of the verification must be performed manually, a formal design technique can facilitate the analysis by providing a clear statement of the design. Several such design techniques are in current use. Top Down Design proposed by Harlan Mills of IBM [MILL70], Structured Design introduced by L. Constantine [YOUR79], and the Jackson Method [JACK75] are examples of manual techniques. The Design Expression and Configuration Aid (DECA) [CARP75], the Process Design Language [CAIN75], Higher Order Software [HAMI76], and SPECIAL [ROUB76] are examples of automated design systems or languages that support automated design analysis and consistency checking.

1.3 The Construction Stage

Actual execution of the code with test data occurs during the construction stage of development. Many testing tools and techniques exist for this stage of system development. Code walk-through and code inspection [FAGA76] are effective manual techniques. Static analysis techniques detect errors by analyzing program characteristics such as data flow and language construct usage. For programs of significant size, automated tools are required to perform this analysis. Dynamic analysis, performed as the code actually executes, is used to determine test coverage through various instrumentation techniques. Formal verification or proof techniques may be used on selected code to provide further quality assurance.

During the entire test process, careful control and management of test information is critical. Test sets, test results, and test reports should be cataloged and stored in a database. For all but very small systems, automated tools are required to do an adequate job, for the bookkeeping chores alone become too large to be handled manually. A test driver, test data generation aids, test coverage tools, test results management aids, and report generators are usually required.

When using the design methodologies described in Section 1.2, at the construction stage, programmers are given design specifications from which they can first code individual modules based on the specification, and then integrate these modules into the completed system. Unless the module being developed is a stand-alone program, it will require considerable auxiliary software to exercise and test it. The auxiliary code that sets up an appropriate environment and calls the module is termed a *driver,* whereas code that simulates the results of a routine called by the module is a *stub.* For many modules both stubs and drivers must be written in order to execute a test. However, techniques can be used to decrease the auxiliary software required for testing. For example, when testing is performed incrementally, an untested module is combined with a tested one and the package is then tested as one, thus lessening the number of drivers and/or stubs that must be written. In *bottom-up* testing, an approach in which the lowest level of modules, those that call no other modules, are tested first and then combined for further testing with the modules that call them, the need for writing stubs can be eliminated. However, test drivers must still be constructed for bottom-up testing. A second approach, *top-down* testing, which starts with the executive module and incrementally adds modules that it calls, requires that stubs be created to simulate the actions of called modules that have not yet been incorporated into the system, but eliminates the need for drivers. The testing order should be chosen to coordinate with the development methodology used.

The actual performance of each test requires the execution of code with input data, an examination of the output, and a comparison of the output with the expected results. Since the testing operation is repetitive in nature, with the same code executed numerous times with different input values, the process of test execution lends itself to automation. Programs that perform this function are called *test drivers, test harnesses,* or *test systems.*

The simplest test drivers merely reinitiate the program with various input sets and save each set of output. The more sophisticated test systems, however, accept not only data inputs, but also expected outputs, the names of routines to be executed, values to be returned by called routines, and other parameters. In addition to initiating the test runs, these test systems also compare the actual output with the expected output and issue concise reports of the performance. TPL/2.0 [PANZ78], which uses a test language to describe test procedures, is an example of such a system. As is typical, TPL/2.0, in addition to executing the test, verifying the results, and producing reports, helps the user generate the expected results.

PRUFSTAND [SNEE78] is an example of such a comprehensive test system. It is an interactive system in which data values are either generated automatically or requested from the user as they are needed. PRUFSTAND is representative of integrated tools systems for software testing and is comprised of (1) a preprocessor to instrument the code; a translator to convert the source data descriptors into an internal symbolic test data description table; (2) a test driver to initialize and update the test environment; (3) test stubs to simulate the execution of called modules; (4) an execution monitor to trace control flow through the test object; (5) a result validator; (6) a test file manager; and (7) a postprocessor to manage reports.

A side benefit of a comprehensive test system is that it establishes a standard format for test materials. This standardization is extremely useful for regression testing, which is discussed in Section 1.4. Currently automatic test driver systems are expensive to build and consequently are not in widespread use.

1.4 The Operation and Maintenance Stage

Over 50 percent of the life-cycle costs of a software system are maintenance [ZELK78, EDP81, GAO81]. As the system is used, it often requires modification either to correct errors or to augment its original capabilities. After each modification, the system must be retested. Such retesting activity is termed *regression testing*. Usually only those portions of the system affected by the modifications need to be retested. However, changes at a given level will necessitate retesting and reverifying products, and updating documentation at all levels below it. For example, a change at the design level requires design reverification, and unit retesting and subsystem and system retesting at the construction level. During regression testing, test cases generated during system development are reused or used after appropriate modifications. Since the materials prepared during development will be reused during regression testing, the quality of the test documentation will affect the cost of regression testing. If test data cases have been cataloged and preserved, duplication of effort will be minimized.

2. VALIDATION, VERIFICATION, AND TESTING TECHNIQUES

Much intense research activity is directed toward developing techniques and tools for validation, verification, and testing. At the same time, a variety of other (and sometimes effective) heuristic techniques and procedures have been put into practice. To describe this diverse collection in a coherent and comparative way is difficult. In this survey we try to follow the life-cycle framework set forth above (summarized in Figure 2) and to integrate the great body of testing heuristics used in practice with the more recent research ideas.

2.1 Testing Fundamentals

Before discussing particular testing methodologies, it is useful to examine testing and its limitations. The objects that we test are the elements that arise during the development of software. These include code modules, requirements and design specifications, data structures, and any other objects necessary for the correct development and implementation of software. We often use the term "program" in this survey to refer to any object that may be conceptually or actually executed. Thus, because design or requirements specifications can be conceptually executed (the flow of the input can be followed through the steps

defined by the specifications to produce a simulated output), remarks directed toward "programs" have broad application.

We view a program as a representation of a function. The function describes the relationship of an input element (called a *domain element*) to an output element (called a *range element*). The testing process is then used to ensure that the program faithfully realizes the function. The essential components of a program test are the program in executable form, a description of the expected behavior, a way of observing program behavior, a description of the functional domain, and a method of determining whether the observed behavior conforms with the expected behavior. The testing process consists of obtaining a valid value from the functional domain (or an invalid value from outside the functional domain, if we are testing for robustness), determining the expected behavior, executing the program and observing its behavior, and finally comparing that behavior with the expected behavior. If the expected and the actual behavior agree, we say that the test instance has succeeded; otherwise, we say that the test instance has uncovered an error.

Of the five necessary components in the testing process, it is frequently most difficult to obtain the description of the expected behavior. Consequently, ad hoc methods often must be used, including hand calculation, simulation, and alternate solutions to the same problem. Ideally, we would construct an *oracle,* a source which, for any given input description, can provide a complete description of the corresponding output behavior.

We can classify program test methods into *dynamic analysis* and *static analysis* techniques. Dynamic analysis requires that the program be executed, and hence follows the traditional pattern of program testing, in which the program is run on some test cases and the results of the program's performance are examined to check whether the program operated as expected. Static analysis, on the other hand, does not usually involve actual program execution (although it may involve some form of conceptual execution). Common static analysis techniques include such compiler tasks as syntax and type checking. We first consider some aspects of static and dynamic analysis within a general discussion of program testing.

A complete verification of a program at any stage in the life cycle can be obtained by performing the test process for every element of the domain. If each instance succeeds, the program is verified; otherwise, an error has been found. This testing method is known as *exhaustive testing* and is the only dynamic analysis technique that will guarantee the validity of a program. Unfortunately, this technique is not practical. Frequently, functional domains are infinite, or even if finite, sufficiently large to make the number of required test instances infeasible.

In order to reduce this potentially infinite exhaustive testing process to a feasible testing process, we must find criteria for choosing representative elements from the functional domain. These criteria may reflect either the functional description or the program structure. A number of criteria, both scientific and intuitive, have been suggested and are discussed.

The subset of elements chosen for use in a testing process is called a *test data set* (*test set* for short). Thus the crux of the testing problem is to find an adequate test set, one large enough to span the domain and yet small enough that the testing process can be performed for each element in the set. Goodenough and Gerhart [GOOD75] present the first formal treatment for determining when a criterion for test set selection is adequate. In their paper, a criterion C is said to be *reliable* if the test sets T_1 and T_2 chosen by C are such that all test instances of T_1 are successful exactly when all test instances of T_2 are successful. A criterion C is said to be *valid* if it can produce test sets that uncover all errors. These definitions lead to the fundamental theorem of testing, which states:

> If there exists a consistent, reliable, valid, and complete criterion for test set selection for a program P and if a test set satisfying the criterion is such that all test instances succeed, then the program P is correct

Unfortunately, it has been shown that there is no algorithm to find consistent,

reliable, valid, and complete test criteria [HOWD76]. This confirms the fact that testing, especially complete testing, is a very difficult process. As we shall see, there is no one best way to generate test data or to ensure best coverage, even heuristically. Combinations of various techniques can increase our confidence in the quality of the software being tested. These combinations depend heavily on the particular instance of the problem.

Probably the most discouraging area of research is that of testing theory, precisely because results such as these abound, showing that testing can never guarantee correctness. Many of the sophisticated techniques that have been recently developed are proving intractable in practical applications. At the same time, many of the heuristics in practice, while often successfully used, do not have a solid theoretical basis from which they can be generalized or validated. Still the importance of the validation and verification process in software development cannot be overstated. By using a variety of techniques and gaining a thorough understanding of the implications and limitations of these techniques, we can increase our confidence in the quality of the software.

2.2 General Techniques

Some techniques are used at many stages. These include traditional informal methods such as desk checking as well as disciplined techniques such as structured walk-throughs and inspections. Proof-of-correctness research is now beginning to produce practical and effective tools and techniques that can be made part of each stage of software development. Moreover, there are other tools, such as simulation, that, although not specific to testing, are highly useful in the validation, verification, and testing process.

2.2.1 Traditional Manual Methods

Desk checking, going over a program by hand while sitting at one's desk, is the most traditional means for analyzing a program, and forms the foundation for the more disciplined techniques of walk-throughs, inspections, and reviews. Requirements, design specifications, and code must always be hand analyzed as it is developed. To be effective this analysis must be careful and thorough. In most instances, this, as well as all other desk checking, is used more as a debugging technique than as a testing technique. Since seeing one's own errors is difficult, it is more effective if a second party does the desk checking. For example, two programmers trading listings and reading each other's code is often more productive than each reading his own. This approach still lacks the group interaction and insight present in formal walk-throughs, inspections, and reviews.

Another method of increasing the overall quality of software production is peer review, the reviewing of programmer's code by other programmers [MYER79]. The management can set up a panel that reviews sample code on a regular basis for efficiency, style, adherence to standards, etc., and that provides feedback to the individual programmer. Project leaders or chief programmers can maintain a notebook that contains both required "fixes" and revisions to the software and an index indicating the original programmer or designer. In a "chief programmer team" [BAKE72] environment, the librarian can collect data on programmer runs, error reports, etc., and act as a review board or pass the information on to a separate peer review panel.

2.2.2 Walk-Throughs, Inspections, and Reviews

Walk-throughs and inspections are formal manual techniques that are a natural evolution of desk checking. While both techniques share a common philosophy and similar organization, they are quite different in execution. Furthermore, although they evolved from the simple desk check discipline of the single programmer, the disciplined procedures of both are aimed at removing the major responsibility for verification from the programmer.

Both walk-throughs and inspections require a team, usually directed by a moderator and including the software developer. The remaining three to six members and the moderator should not be directly in-

volved in the development effort. Both techniques are based on a reading of the product (e.g., requirements, specifications, or code) in a formal meeting environment with specific rules for evaluation. The difference between inspection and walk-through lies in the conduct of the meeting.

Inspection involves a step-by-step reading of the product, with each step checked against a predetermined list of criteria. (These criteria include checks for historically common errors, adherence to programming standards, and consistency with program specifications.) Guidance for developing the test criteria can be found in MYER79, FAGA76, and WEIN71. Usually the developer narrates the reading of the product and finds many errors just by the simple review act of reading aloud. Others errors, of course, are determined as a result of the discussion with team members and by applying the test criteria.

Walk-throughs differ from inspections in that the programmer does not narrate a reading of the product by the team. A team leader, either the developer or another person, provides test data and leads the team through a manual simulation of the system. The test data are walked through the system, with intermediate results kept on a blackboard or paper. The test data should be kept simple, given the constraints of human simulation. The purpose of the walk-through is to encourage discussion, not just to complete the system simulation on the test data. Most errors are discovered by questioning the developer's decisions at various stages, rather than by examining the test data.

At the problem definition stage, either walk-through or inspection can be used to determine whether the requirements satisfy the testability and adequacy measures of this stage in development. If formal requirements have been developed, formal methods, such as correctness techniques, may be applied to ensure adherence to the quality factors.

Walk-throughs or inspections should be performed again at the preliminary and detailed design stages, especially in examining the testability and adequacy of module and module interface designs. Any changes that result from these analyses will cause at least a partial repetition of the verification at the problem definition and earlier design stages, with an accompanying reexamination of the consistency between stages.

Finally, the walk-through or inspection procedures should be performed on the code produced during the construction stage. Each module should be analyzed both separately and then as an integrated part of the finished software.

Design reviews and audits are commonly performed as stages in software development. The Department of Defense has developed a standard audit and review procedure [MILS76] based on hardware procurement regulations. The process is representative of the use of formal reviews and includes several stages (detailed in the glossary).

2.2.3 Proof-of-Correctness Techniques

The most complete static analysis technique is proof of correctness. At an informal level, proof-of-correctness techniques reduce to the sort of step-by-step reasoning involved in an inspection or a walk-through. At a more formal level, the machinery of mathematical logic is brought to bear on the problem of proving that a program meets its specification.

Proof techniques as methods of validation have been used since von Neumann's time. These techniques usually consist of validating the consistency of an output "assertion" (specification) with respect to a program (or requirements or design specification) and an input assertion (specification). In the case of programs, the assertions are statements about the program's variables. If it can be shown that executing the program causes the output assertion to be true for the possibly changed values of the program's variables whenever the input assertion is true for particular values of variables, then the program is "proved." To be completely sure that a program is correct, the programmer must also prove that the program terminates. Normally, the issue of termination is handled separately.

There are two approaches to proof of correctness: formal proof and informal proof. In order to obtain formal proofs, a

mathematical logic must be developed with which one can "talk" about programming language objects and can express the notion of computation. Two approaches have been taken in designing such logics: (1) to employ mathematical logic with a natural notion of computation, essentially keeping the two separate [FLOY67]; and (2) to tightly integrate the computation aspects of programming languages with the static, mathematical aspects of programming languages [CONS78, PRAT77]. Because of the computational power of most programming languages, the logic used to verify programs is normally not decidable; that is, there is no algorithm to determine the truth or falsity of every statement in the logic.

Most recent research in applying proof techniques to verification has concentrated on programs. The techniques apply, however, equally well to any level of the development life cycle where a formal representation or description exists. The GYPSY [AMBL78] and HDM [ROBI79, NEUM75] methodologies use proof techniques throughout the development stages. For example, HDM has as a goal the formal proof of each level of development. Good summaries of program proving and correctness research are given in KING76 and APT81.

Since formal mathematical techniques grow rapidly in complexity, heuristic procedures for proving programs formally are essential. Unfortunately, these are not yet well enough developed to allow the formal verification of a large class of programs. In the absence of efficient heuristics, some approaches to verification require that the programmer provide information interactively to the verification system order to complete the proof. Examples include AFFIRM [GERH80], the Stanford PASCAL Verifier [LUCK79], and PL/CV [CONS78]. Such provided information may include facts about the program's domain and operators or facts about the program's intended function.

Informal proof techniques follow the logical reasoning behind the formal proof techniques but without the formal details. Often the less formal techniques are more palatable to the programmers because they are intuitive and not burdened with mathematical formalism. The complexity of informal proof ranges from simple checks, such as array bounds not being exceeded, to complex logic chains showing noninterference of processes accessing common data. Programmers are always using informal proof techniques; if they make the techniques explicit, it would require the same resource investment as following a discipline such as structured walk-through.

Notwithstanding the substantial research efforts in developing useful proof-of-correctness systems, there has been dispute concerning the ultimate utility of automated correctness proving as a useful tool of verification and validation [DEMI79, DIJK78]. It is unlikely that this dispute will be quickly settled, but it is likely that proof-of-correctness techniques will continue to play a role in the validation and verification process.

2 2 4 Simulation

Simulation is a broad term. In a sense any validation technique that does not involve actual execution "simulates" the execution in some fashion. All of the techniques described above thus use simulation by this very broad definition. Even if we employ a more narrow definition, that simulation is the use of an executable model to represent the behavior of an object, simulation, as we shall show, is still a powerful tool for testing.

Simulation is most often employed in real-time systems development where the "real-world" interface is critical and integration with the system hardware is central to the total design. There are, however, many non-real-time applications in which simulation is a cost-effective verification and test data generation technique.

Several models must be developed to use simulation as a verification tool. Verification is performed by determining, with use of simulation, whether the model of the software behaves as expected on models of the computational and external environments.

To construct a model of the software for a particular stage in the development life cycle, one must develop a formal representation of the product at that stage compatible with the simulation system. This representation may consist of the formal re-

quirements specification, the design specification, or the actual code, depending on the stage, or it may be a separate model of the program behavior. If a different model is used, then the developer will need to demonstrate and verify that the model is a complete, consistent, and accurate representation of the software at the stage of development being verified.

After creating the formal model for the software, the developer must construct a model of the computational environment in which the system will operate. This model will include, as components, representations of the hardware on which the system will be implemented and of the external demands on the total system. This model can be largely derived from the requirements, with statistical representations developed for the external demand and the environmental interactions.

Simulating the system at the early development stages is the only means of predicting the system behavior in response to the eventual implementation environment. At the construction stage, since the code is sometimes developed on a host machine quite different from the target machine, the code may be run on a simulation of the target machine under interpretative control.

Simulation also plays a useful role in determining the performance of algorithms. While this is often directed at analyzing competing algorithms for cost, resource, or performance trade-offs, the simulation of algorithms on large data sets also provides error information.

2.3 Test Data Generation

Test data generation is a critical step in testing. Test data sets must not only contain input to exercise the software, but must also provide the corresponding correct output responses to the test data inputs. Thus the developing of test data sets involves two aspects: the selecting of data input and the determining of expected response. Often the second aspect is most difficult, because, although hand calculation and simulation can be used to derive expected output response, such manual techniques become unsatisfactory and insufficient for very large or complicated systems.

One promising direction is the development of executable specification languages and specification language analyzers [SRS79, TEIC77]. These can be used to act as "oracles," providing the responses for the test data sets. Some analyzers such as the REVS system [BELL77] include a simulation capability. An executable specification language representation of a software system is an actual implementation of the design, but at a higher level than the final code. Usually interpreted rather than compiled, it is less efficient, omits certain details found in the final implementation, and is constructed with certain information "hidden." This implementation would be, in Parnas' terms [PARN77], an "abstract program," representing in less detail the final implementation. The execution of the specification language "program" could be on a host machine quite different from the implementation target machine.

Test data can be generated randomly with specific distributions chosen to provide some statistical assurance that the system, after having been fully tested, is error free. This is a method often used in high-density large-scale integrated (LSI) testing. Unfortunately, while errors in LSI chips appear correlated and statistically predictable, this is not true of software. Until recently, the domains of programs were far more intractable than those occurring in hardware. This gap is closing with the advances in very large-scale integration (VLSI).

Given the apparent difficulty of applying statistical tests to software, test data are derived in two global ways, often called "black box," or functional, analysis and "white box," or structural, analysis. In functional analysis, the test data are derived from the external specification of the software behavior with no consideration given to the internal organization, logic, control, or data flow. One such technique, design-based functional analysis [HOWD80a], includes examination and analysis of data structure and control flow requirements and specifications throughout the hierarchical decomposition of the system during the design. In a complementary fashion, tests derived from structural analysis depend almost completely on the internal log-

ical organization of the software. Most structural analysis is supported by test coverage metrics, such as path coverage and branch coverage. These criteria provide a measure of completeness of the testing process.

2.4 Functional Testing Techniques

The most obvious and generally intractable functional testing procedure is exhaustive testing. As was described earlier, only a fraction of programs can be exhaustively tested. All programs that can be exhaustively tested can be replaced with a table lookup procedure. Generally, the domain of a program is usually infinite or infeasibly large and cannot be used as a test data set. Since exhaustive testing is not possible, characteristics of the input domain are examined for ways of deriving a sufficiently representative test data set to fully exercise the system.

Test data must be derived from an analysis of the functional requirements and include representative elements from all the variable domains. These data should include both valid and invalid inputs. Generally, data in test data sets based on functional requirements analysis can be characterized as *extremal*, *nonextremal*, or *special*, depending on the source of their derivation. The properties of these elements may be simple *values*, or for more complex data structures they may include such attributes as *type* and *dimension*.

2.4.1 Boundary Value Analysis

The problem of deriving test data sets is to partition the program domain so that input data sets that span the partition can be determined. There is no direct, easily stated procedure for forming this partition. The partitioning depends on the requirements, the program domain, and the creativity and problem understanding of the programmer. This partitioning, however, should be performed throughout the development life cycle.

At the problem definition, the overall functional requirements provide a coarse partitioning. At the design stage, additional functions define the separate modules and thus allow a refinement of the partition. Finally, at the coding stage, submodules implementing the design modules introduce further refinements. The use of a top-down testing methodology allows each of these refinements to be used to construct functional test cases at the appropriate level. The higher level refinements can be used to guide integration testing over the modules represented by lower level refinements. HOWD78 and MYER79 give examples and further guidance.

Once the program domain is partitioned into input classes, functional analysis can be used to derive test data sets. Chosen test data should lie both inside each input class and at the boundary of each class. Output classes should also be covered by selecting test data input that causes output at each class boundary and within each class. These data are the extremal and nonextremal test sets. Determination of these test sets is often called *boundary value analysis* or *stress testing*.

The boundary values chosen depend on the nature of the data structures and the input domains. Consider the following FORTRAN example:

```
INTEGER X
REAL A(100, 100)
```

If X is constrained, $a < X < b$, then X should be tested for valid inputs $a + 1$, $b - 1$, and invalid inputs a and b. Invalid input data will test the robustness of the program. The array should be tested as a single element array A(1, 1) and as a full 100×100 array. The array element values A(I, J) should be chosen to exercise the corresponding boundary values for each element. Examples for choosing boundary values in more complex data structures can be found in HOWD78 and MYER79.

Myers suggests that some people have a natural intuition for test data generation [MYER79]. While this ability cannot be completely described or formalized, certain test data seem highly likely to catch errors. Some of these are in the category Howden [HOWD80a] calls *special*; others are certainly boundary values. Zero-input values and input values that drive the outputs to zero are examples. For more complicated data structures, the equivalent null data structure such as an empty list or stack or

a null matrix should be tested. Often the single-element data structure is a good choice. If numeric values are used in arithmetic computations, then the test data should include values that are numerically very close and values that are numerically quite different. Guessing carries no guarantee for success, but neither does it carry any penalty.

2 4 2 Design-Based Functional Testing

The techniques described above derive test data sets from analysis of functions specified in the requirements. Howden has extended functional analysis to functions used in the design process [HOWD80a]. A distinction can be made between requirements functions and design functions. Requirements functions describe the overall functional capabilities of a program, and cannot usually be implemented without the developer first inventing other "smaller functions" to design the program. If one thinks of this relationship as a tree structure, then a requirements function would be represented as a root node, and the "smaller functions," all those functional capabilities corresponding to design functions, would be represented by boxes at the second level in the tree. Implementing one design function may require inventing still other design functions. This successive refinement during top-down design can then be represented as levels in the tree structure, where the $(n + 1)$st-level nodes are refinements or subfunctions of the nth-level functions.

To utilize design-based functional testing, the functional design trees as described above should be constructed. The functions included in the design trees must be chosen carefully with the most important selection criteria being that the function be accessible for independent testing. It must be possible to find a test data set that tests the function, to derive the expected values for the function, and to observe the actual output computed by the code implementing the function.

If top-down design techniques are followed, each of the functions in the functional design tree can be associated with the final code used to implement that function. This code may consist of one or more procedures, parts of a procedure, or statements. Design-based functional testing requires that the input and output variables for each design function be completely specified. Given these multiple functions to analyze, test data generation can proceed as described in the boundary value analysis discussion above. Extremal, nonextremal, and special-values test data should be selected for each input variable. Test data should also be selected to generate extremal, nonextremal, and special-output values.

2 4 3 Cause–Effect Graphing

Cause-effect graphing [MYER79] is a technique for developing test cases for programs from the high-level specifications. For example, a program that has specified responses to eight characteristic stimuli (called causes) has potentially 256 "types" of input (i.e., those with characteristics 1 and 3, those with characteristics 5, 7, and 8, etc.). A naive approach to test case generation would be to try to generate all 256 types. A more sophisticated approach is to use the program specifications to analyze the program's effect on the various types of inputs.

The program's output domain can be partitioned into various classes called "effects." For example, inputs with characteristic 2 might be subsumed by (i.e., cause the same effect as) those with characteristics 3 and 4. Hence, it would not be necessary to test inputs with just characteristic 2 and also inputs with characteristics 3 and 4. This analysis results in a partitioning of the causes according to their corresponding effects.

After this analysis, the programmer can construct a limited-entry decision table from the directed graph reflecting these dependencies (i.e., causes 2 and 3 result in effect 4; causes 2, 3, and 5 result in effect 6; and so on), reduce the decision table in complexity by applying standard techniques [METZ77], and choose test cases to exercise each column of the table. Since many aspects of the cause-effect graphing can be automated, it is an attractive tool for aiding in the generation of functional test cases.

2.5 Structural Testing Techniques

Unlike functional testing, which was concerned with the function the program performed and did not deal with how the function was implemented, structural testing is concerned with testing its implementation. Although used primarily during the coding phase, structural testing should be used in all phases of the life cycle where the software is represented formally in some algorithmic, design, or requirements language. The intent of structural testing is to find test data that will force sufficient coverage of the structures present in the formal representation. In order to determine whether the coverage is sufficient, it is necessary to have a structural coverage metric. Thus the process of generating tests for structural testing is sometimes known as *metric-based test data generation.*

Metric-based test data generation can be divided into two categories by the metric used: coverage-based testing and complexity-based testing. In the first category, a criterion is used that provides a measure of the number of structural units of the software which are fully exercised by the test data sets. In the second category, tests are derived in proportion to the software complexity.

2.5.1 Coverage-Based Testing

Most coverage metrics are based on the number of statements, branches, or paths in the program that are exercised by the test data. Such metrics can be used both to evaluate the test data and to aid in the generation of the test data.

Any program can be represented by a graph. The nodes represent statements or collections of sequential statements, and the lines or edges represent the control flow. A node with a single exiting edge to another node represents a sequential code segment. A node with multiple exiting edges represents a branch predicate or a code segment containing a branch predicate as the last statement.

As an example of the representation of a program by a graph, consider the bubble sort program of Figure 3 (from an example due to PAIG77) and its associated program graph shown in Figure 4.

```
1  SUBROUTINE BUBBLE (A, N)
2    BEGIN
3    FOR I = 2 STEPS 1 UNTIL N DO
4      BEGIN
5      IF A(I) GE A(I - 1) THEN GOTO NEXT
6      J = I
7    LOOP: IF J LE 1 THEN GOTO NEXT
8      IF A(J) GE A (J - 1) THEN GOTO NEXT
9      TEMP = A(J)
10     A(J) = A(J - 1)
11      A(J - 1) = TEMP
12     J = J - 1
13     GOTO LOOP
14   NEXT· NULL
15     END
16   END
```

Figure 3. A bubble sort program. (Adapted from PAIG77, *IEEE Transactions on Software Engineering* **SE-3,** 6 (Nov. 1977), 387, with permission of the IEEE.)

On a particular set of data, a program will execute along a particular *path*, where certain *branches* are taken or not taken, depending on the evaluation of branch predicates. Any program path can be represented by a sequence, possibly with repeating subsequences (when the program has backward branches), of edge names from the program graph. These sequences are called *path expressions.* Each path or each data set may vary, depending on the number of *loop iterations* executed. A program with variable loop control may have effectively an infinite number of paths, and hence an infinite number of path expressions.

To test the program structure completely, the test data chosen should ideally cause the execution of all paths. But because some, possibly many, paths in a program are not finite, they cannot be executed under test conditions. Since complete coverage is not possible in general, metrics have been developed that give a measure of the quality of test data based on its proximity to this ideal coverage. Path coverage determination is further complicated by the existence of *infeasible paths*, that, owing to inadvertent program design, are never executed, no matter what data are used. Automatic determination of infeasible paths is generally difficult if not impossible. A main theme in structured top-down design [DIJK72, JACK75, YOUR79] is to construct modules that are simple and of low com-

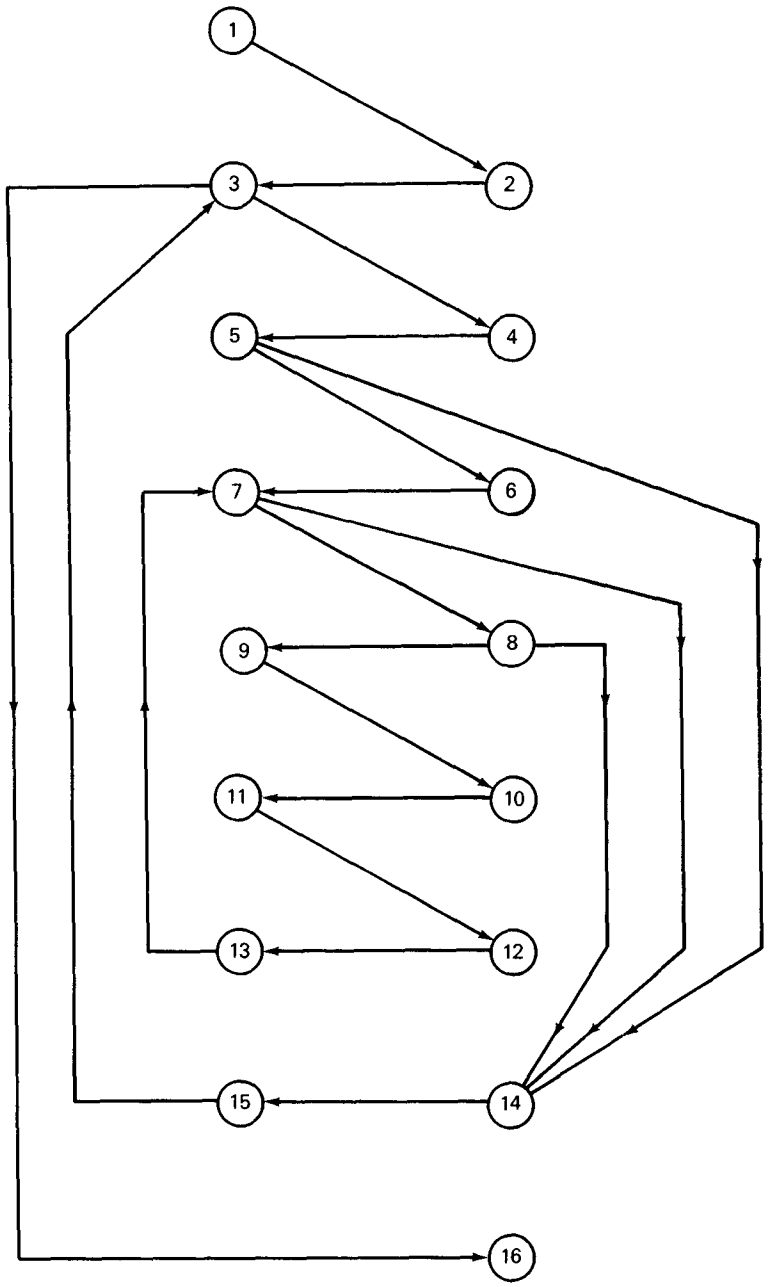

Figure 4. Control-flow graph for the program in Figure 3 (Adapted from PAIG77, *IEEE Transactions on Software Engineering* **SE-3,** 6 (Nov. 1977), 389, with permission of the IEEE)

plexity so that all paths, excluding loop iteration, may be tested and that infeasible paths may be avoided. Of course, during integration testing when simple modules are combined into more complex modules, paths will cross modules and infeasible paths may again arise. The goal is to maintain simple structure at all levels of integration, therefore maximizing path coverage.

All techniques for determining coverage metrics are based on graph requirements of programs. A variety of metrics exist ranging from simple-statement coverage to full-path coverage. There have been several attempts to classify these metrics [MILL77]; however, new variations appear so often that such attempts are not always successful. We discuss the major ideas without attempting to cover all the variations.

The simplest metric measures the percentage of statements executed by all the test data. Since coverage tools collect data about which statements have been executed (as well as about the percentage of coverage), results can guide the programmer in selecting test data to ensure complete coverage. To apply the metric, the programmer instruments the program or module either by hand or by a preprocessor, and then uses either a postprocessor or manual analysis of the results to find the level of statement coverage. Finding an efficient and complete test data set that satisfies this metric is more difficult. Branch predicates that send control to omitted statements can, when examined, help determine which input data will cause execution of those omitted statements.

Examination of the program's actions on the test set, $S_1 = \{A(1) = 5, A(2) = 3, N = 2\}$ (Figure 3), demonstrates that 100 percent statement coverage is reached. This metric, however, is not strong enough. A slight change in the example program (replacing the greater or equal test by an equality test) results in an incorrect program and an error that the test set does not uncover.

A slightly stronger metric measures the percentage of *segments* executed under the application of all test data. A segment in this sense corresponds to a *decision-to-decision path* (*dd path*) [MILL77]. It is a portion of a program path beginning with the execution of a branch predicate and including all statements up to the evaluation (but not execution) of the next branch predicate. In the example of Figure 4, the path including statements 8, 9, 10, 11, 12, 13 is a segment. Segment coverage clearly guarantees statement coverage. It also covers branches with no executable statements, as in the case in an IF-THEN-ELSE with no ELSE statement; coverage still requires data, causing the predicate to be evaluated as both true and false, and segment coverage guarantees that both have been checked. Techniques similar to those used for statement coverage are used for applying the metric and deriving test data.

Returning to the example program, the test data set, S_1, proposed earlier does not cover the two segments with no executable statements (segments beginning at nodes 5 and 8). The set

$$S_2 = \{\{A(1) = 5, A(2) = 3, A(3) = 3, N = 3\}, \{A(1) = 3, A(2) = 5, N = 2\}\}$$

yields 100 percent segment coverage, but still does not uncover the error introduced by replacing greater or equal by equal.

Often a loop construct is improperly used. An improper termination may result when the loop predicate is not initially satisfied. Thus, the next logical step is to strengthen the metric by requiring separate coverage for both the exterior and interior of loops. Since segment coverage only requires that both branches from a branch predicate be taken, the situation can arise that test sets always execute the loop body at least once (satisfies internal test) before the exiting branch is traversed (external test satisfied). To ensure that a test data set contains data that requires the exiting branch to be taken without executing the loop body, segment coverage is strengthened so as to require that external tests be performed without loop body execution. This metric requires more paths to be covered than does segment coverage, whereas segment coverage requires more paths to be covered than does statement coverage.

In the example, adding $\{A(1) = 3, N = 1\}$ to the test data set S_2 gives a test

set, S_3, that forces execution of both the interior and exterior of the FOR loop. The single element array ensures that the loop controlling predicate is tested without execution of the loop body.

Variations on the loop and segment metric include requiring at least k interior iterations per loop or requiring that all 2^n combinations of Boolean variables be applied for each n-variable predicate expression. The latter variation has led to a new path-testing technique called *finite-domain testing* [WHIT78].

Automated tools for instrumenting and analyzing the code have been available for a few years [MILL75, OSTE76, LYON74, RAMA74, MAIT80]. These tools are generally applicable to most of the coverage metrics described above. Automating test data generation, however, is less advanced. Often test data are generated by iteratively using analyzers, and then applying manual methods for deriving tests. A promising but expensive way to generate test data for path testing is through the use of symbolic executors [BOYE75, KING76, CLAR77, HOWD77]. The use of these tools is discussed further in Section 2.7. Even though any particular structural metric may be satisfied, there is still no guarantee that software is correct. As discussed in Section 2.1, the only method of ensuring that the testing is complete is to test the program exhaustively. None of the above coverage metrics, nor any proposed coverage metrics, guarantees exhaustive testing. The choice of which coverage metric to use must be guided by the resources available for testing. A coverage metric that forces more paths to be tested in order to achieve the same coverage as a simplier metric is more expensive to use because more test cases must be generated. The last few errors uncovered can cost several orders of magnitude more than the first error uncovered.

2.5.2 Complexity-Based Testing

Several complexity-based metrics have been proposed recently. Among these are cyclomatic complexity [MCCA76], Halstead's metrics [HALS77], and Chapin's software complexity measure [CHAP79]. These and many other metrics are designed to analyze the complexity of software systems. Although these metrics are valuable new approaches to the analysis of software, most are unsuited, or have not been applied to the problem of testing. The McCabe metrics are the exception.

McCabe actually proposed three metrics: *cyclomatic*, *essential*, and *actual complexity*. All three are based on a graphical representation of the program being tested. The first two metrics are calculated from the program graph, while the third metric is calculated at run time.

McCabe defines cyclomatic complexity by finding the graph theoretic "basis set." In graph theory, there are sets of linearly independent program paths through any program graph. A maximal set of these linearly independent paths, called a "basis set," can always be found. Intuitively, since the program graph and any path through the graph can be constructed from the basis set, the size of this basis set should be related to the program complexity. From graph theory, the cyclomatic number of the graph, $V(G)$, is given by

$$V(G) = e - n + p$$

for a graph G with number of nodes n, edges e, and connected components p. The number of linearly independent program paths though a program graph is $V(G) + p$, a number McCabe calls the cyclomatic complexity of the program. Cyclomatic complexity, $CV(G)$, where

$$CV(G) = e - n + 2p,$$

can then be calculated from the program graph. In the graph of Figure 4, $e = 19$, $v = 16$, and $p = 1$. Thus $V(G) = 4$ and $CV(G) = 5$.

A proper subgraph of a graph G is a collection of nodes and edges such that, if an edge is included in the subgraph, then both nodes it connects in the complete graph G must also be in the subgraph. Any flow graph can be reduced by combining sequential single-entry, single-exit nodes into a single node. Structured constructs appear in a program graph as proper subgraphs with only one single-entry node whose entering edges are not in the subgraph, and with only one single-exit

node, whose exiting edges are also not included in the subgraph. For all other nodes, all connecting edges are included in the subgraph. This single-entry, single-exit subgraph can then be reduced to a single node.

Essential complexity is a measure of the "unstructuredness" of a program. The degree of essential complexity depends on the number of these single-entry, single-exit proper subgraphs containing two or more nodes. There are many ways in which to form these subgraphs. For a straight-line graph (no loops and no branches), it is possible to collect the nodes and edges to form from 1 to $v/2$ (v = number of nodes) single-entry, single-exit subgraphs. Hecht and Ullman [HECH72] have a simple algorithm that is guaranteed to find the minimum number of such subgraphs in a graph. Figure 5 is an example of a program graph with single-entry, single-exit proper subgraphs identified from Hecht and Ullman's algorithm. The nodes in the four proper subgraphs are {1, 2}, {3, 4, 5, 6, 16}, {7, 8, 9, 10, 11, 12, 13}, and {14, 15}.

Let m be the minimum number calculated from Hecht and Ullman's algorithm. The essential complexity EV(G) is defined as

$$\mathrm{EV}(G) = \mathrm{CV}(G) - m.$$

The program graph for a program built with structured constructs will generally be decomposable into subgraphs that are single entry, single exit. The minimum number of such proper subgraphs (calculated from Hecht and Ullman's algorithm) is CV(G) − 1. Hence, the essential complexity of a structured program is 1. The program of Figure 3 has essential complexity of 1 indicating that the program is structured.

Actual complexity, AV, is the number of independent paths actually executed by a program running on a test data set. AV is always less than or equal to the cyclomatic complexity and is similar to a path coverage metric. A testing strategy would be to attempt to drive AV closer to CV(G) by finding test data which cover more paths or by eliminating decision nodes and reducing portions of the program to in-line code. There exist tools [MAIT80] to calculate all three McCabe metrics.

2.6 Test Data Analysis

After the construction of a test data set, it is necessary to determine the "goodness" of that set. Simple metrics like statement coverage may be required to be as high as 90–95 percent. It is much more difficult to find test data providing 90 percent coverage under the more complex coverage metrics. However, it has been noted [BROW73] that methods based on the more complex metrics with lower coverage requirements have uncovered as many as 90 percent of all program faults.

2.6.1 *Statistical Analyses and Error Seeding*

The most common type of test data analysis is statistical. An estimate of the number of errors in a program can be obtained by analyzing of errors uncovered by the test data. In fact, as we shall see, this leads to a dynamic testing technique.

Let us assume that there are some number of errors E in the software being tested. We would like to two things: a maximum likelihood estimate for the number of errors and a level-of-confidence measure on that estimate. Mills developed a technique [MILL72] to "seed" known errors into the code so that their placement is statistically similar to that of actual errors. The test data are then applied, and the number of known seeded errors and the number of original errors uncovered is determined. If one assumes that the statistical properties of the seeded and unseeded errors are the same (i.e., that both kinds of errors are equally findable) and that the testing and seeding are statistically unbiased, then the maximum-likelihood estimator for E is given by

$$\text{estimate } E = IS/K$$

where S is the number of seeded errors, K is the number of discovered seeded errors, and I is the number of discovered unseeded errors. This estimate obviously assumes that the proportion of undetected errors is very likely to be the same for the seeded and original errors. This assumption is open to criticism [SCHI78] since many errors left after the debugging stage are very subtle, deep logical errors [DEMI78], which are not

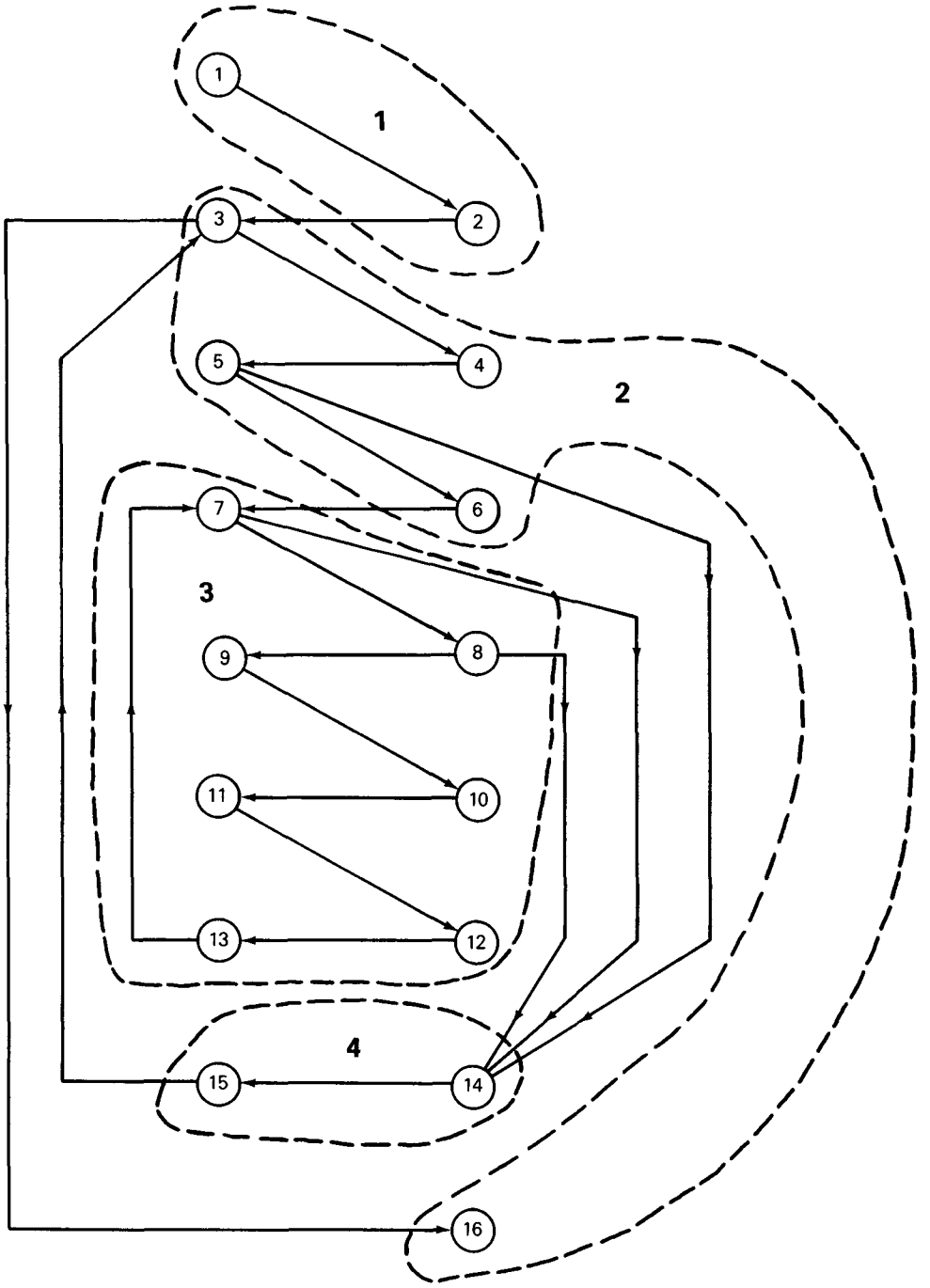

Figure 5. Example from Figure 4 with subgraphs identified.

statistically independent and are likely to be quite different from the seeded errors.

Mills developed confidence levels for his techniques, which are revised and discussed in TAUS77. A further and perhaps more complete examination of confidence levels is described in DURA81a. A strategy for using this statistical technique in dynamic testing is to monitor the maximum likelihood estimator, and to perform the confidence-level calculation as testing progresses. If the estimator becomes high relative to the number of seeded errors, then it is unlikely that a desirable confidence level can be obtained. The seeded errors should be removed and the testing resumed. If the number of real errors discovered remains small (ideally, remains zero) as the number of seeded errors uncovered approaches the total number seeded, then our confidence level increases.

Schick and Wolverton [SCHI78] and others have described a technique of using two people to test the software, using one person's discovered errors as the "seeded" errors and then applying the estimator to the second person's results. But it is difficult to make the two people's testing procedures sufficiently different so that the overlap in their uncovered errors is small; as the overlap increases, confidence of the estimation must decrease.

Tausworthe [TAUS77] discusses a method for seeding errors that has some hope of imitating the distribution of the actual errors. He suggests randomly choosing lines at which to insert the error, and then making various different modifications to the code, introducing errors. The modifications of the code are similar to those used in mutation testing as described below. Duran and Wiorkowski [DURA81a] suggest using errors detected during preliminary testing as seed errors for this technique. In either case, again, success depends on the detected errors having the same probability of detection as the undiscovered errors, which is not likely.

2 6 2 Mutation Analysis

A new method of determining the adequacy of test data sets has been developed by DeMillo, Lipton, and Sayward and is called mutation analysis [DEMI78]. As above, the program that is to be tested is seeded with errors. Several mutants of the original program are generated. Each is created by introducing different errors or sets of errors into the original program. The program and its mutants are then run interpretively on the test set.

The set of mutants must be held to a manageable size. First, consider the "competent programmer assumption," stating that an incorrect program will not differ much from the desired program. That is, a competent programmer will not make a massive number of errors when writing a program. Second, consider the "coupling effect," the conjecture that tests that uncover simple errors will also uncover deeper and more complex errors.

These two assumptions greatly simplify the construction of program mutations. To determine the adequacy of test sets, we introduce a mutation score ms(P, T) defined as

$$\text{ms}(P, T) = |\text{DM}(P, T)| / |\text{M}(P) - \text{E}(P)|,$$

where P is a program, T is a test set, M(P) is some finite set of mutant programs of the language, E(P) is the set of functionally equivalent programs to P in M(P), and DM(P, T) is the set of programs in M(P) differentiated from P by the test set T. If the construction of mutants is correctly chosen (i.e., the finite set of program mutations is appropriately constructed), then as the mutation score, ms(P, T), approaches 1, the adequacy of the test set T increases (and T uncovers more errors).

The construction of the set of mutations is crucial to the success of the technique. The mutant set is obtained from P by modifying single statements of the program in order to reflect probable errors. Since each element of the finite set of program mutations differs from P in only one statement and since variable names may be changed in order to construct elements of the set of mutations, the size of M(P) is bounded by a quadratic function of the length of P.

The mutation analysis method of determining the adequacy of test sets includes both branch coverage and statement coverage metrics as special cases. Over the last two years, the method has been run on a number of FORTRAN and COBOL programs ranging from a few lines in length to

production programs of 1700 lines in length. Test sets with mutation scores of 0.95 or higher were experimentally shown to be adequate in that additional errors were not discovered with subsequent use of the programs [ACRE80].

It must be stressed that mutation analysis rests on two assumptions: that the program is "nearly correct" (a consequence of the competent programmer hypothesis) and that test sets which uncover single errors are also effective in uncovering multiple errors (the coupling effect hypothesis). Both of these assumptions have been experimentally validated over a fairly large range of programs [ACRE80].

Recently Howden [HOWD81a] developed a new test completeness metric that is stronger than branch coverage, but weaker than mutant coverage. Derived from the ideas on design-based functional testing, the metric depends either on coverage of functions computed by a program, *parts* of the program, or by *parts* of statements in the program. This method is less costly than mutation analysis, but much more effective than branch coverage.

2.7 Static Analysis Techniques

As we stated at the outset, analytical techniques can be categorized as dynamic or static. Dynamic activity, such as the application and analysis of test data, usually involves the actual execution of code, whereas static analysis usually does not. Many of the general techniques discussed above, such as formal proof techniques and inspections, are static analysis techniques. Static analysis is part of any testing technique, since it must be used in analysis that derives test data, calculates assertions, or determines instrumentation breakpoints. But the actual verification must be achieved through dynamic testing. The line between static and dynamic analysis is not always easily drawn. For example, proof-of-correctness techniques and symbolic execution both "execute" code, but usually not in a real environment.

Most static analysis is performed by parsers and associated translators residing in compilers. Depending upon the sophistication of the parser, it uncovers errors ranging in complexity from ill-formed arithmetic expressions to complex type-incompatibilities. In most compilers, the parser and translator are augmented with additional capabilities that allow activities useful for producing quality software, such as code optimization, listing of variable names, and pretty printing. Preprocessors are also frequently used in conjunction with the parser. These may perform activities such as allowing "structured programming" in an unstructured programming language, checking for errors such as mismatched common areas, and checking for module interface incompatibilities. The parser may also serve in a policing role. Thus, by using static analysis the parser can enforce coding standards, monitor quality of code, and check adherence to programming standards (standards such as FORTRAN77 [ANSI78].

2 7 1 Flow Analysis

Data-flow and control-flow analysis are similar in many ways. Both are based upon graphical representation. In control-flow analysis, the program graph has nodes, representing a statement or segment, that possibly end in a branch predicate. The edges represent the allowed flow of control from one segment to another. The control-flow graph is used to analyze the program behavior, to locate instrumentation breakpoints, to identify paths, and to perform static analysis activities. In data-flow analysis, graph nodes usually represent single statements, while the edges still represent the flow of control. Nodes are analyzed to determine the transformations made on program variables. Data-flow analysis is used to discover program anomalies such as undefined or unreferenced variables. Data-flow analysis was used by Cocke and Allen [ALLE74, ALLE76] to do global program optimization.

Data-flow anomalies are more easily found than resolved. Consider the following FORTRAN code segment:

```
SUBROUTINE HYP (A, B, C)
U = 0.5
W = 1/V
Y = A ** W
Y = E ** W
Z = X + Y
C = Z ** (V)
```

There are several anomalies in this code segment. One variable, U, is defined and never used, while three variables, X, V, and E, are undefined when used. The problem is not in detecting these errors, but in resolving them. It is possible, for instance, that U was meant to be V, E was meant to be B, and the first occurrence of Y on the left of an assignment was a typo for X. There is no answer to the problem of resolution, but data-flow analysis can help to detect the anomalies, including ones more subtle than those above.

In data-flow analysis, we are interested in tracing the behavior of program variables as they are initialized and modified during the program execution. This behavior can be classified according to when a particular variable is *referenced*, *defined*, or *unreferenced* in the program. A variable is referenced when its value is obtained from memory during the evaluation of an expression in a statement. For example, a variable is referenced when it appears on the right-hand side of an assignment statement, or when it appears as an array index anywhere in a statement. A variable is defined if a new value for that variable results from the execution of a statement, as occurs when a variable appears on the left-hand side of an assignment. A variable becomes unreferenced when its value is no longer determinable from the program flow. Examples of unreferenced variables are local variables in a subroutine after exit and FORTRAN DO indices on loop exit.

Data-flow analysis is performed, at each node in the data flow graph, by associating values for tokens (the latter representing program variables) that indicate whether the corresponding variable is referenced, unreferenced, or defined with the execution of the statement represented by that node. If, for instance, the symbols, u, d, r, and l (for null), are used to represent the values of a token, then *path expressions* for a variable (or token) can be generated beginning at, ending in, or for some particular node, yielding, for example, the typical path expression drllllrrllllldllrll. This expression can then be reduced, by eliminating nulls, to drrrdru. Such a path expression contains no anomalies, but the presence of a double nonnull value in an expression, such as . . .dd. . ., indicates a variable defined twice without being referenced, and does identify a potential anomaly. Most anomalies, such as unreferenced followed by referenced or referenced without being defined, can be discovered through analysis of the path expressions.

To simplify the analysis of the flow graph, statements can be combined, as in control-flow analysis, into segments of necessarily sequential statements represented by a single node. Often, however, statements must be represented by more than one node. Consider the expression,

```
IF (X.GT.1) X = X - 1
```

The variable X is certainly referenced in the statement, but it will be defined only if the predicate is true. In such a case, the representation would use two nodes, and the graph would actually represent the code:

```
    IF (X.GT.1) 100, 200
100 X = X - 1
200 CONTINUE
```

Another problem requiring node splitting arises at the last statement of a FORTRAN DO loop, in which case the index variable will become undefined if the loop is exited. The problems introduced by subroutine and function calls can also be resolved using data-flow analysis. Osterweil [OSTE76] and Fosdick [FOSD76] describe the use of data-flow analysis for static analysis and testing.

2 7.2 Symbolic Execution

Symbolic execution is a method of symbolically defining data that forces program paths to be executed. Instead of executing the program with actual data values, the variable names that hold the input values are used as input values.

All branches are taken during a symbolic execution, and the effect of assignments during a symbolic execution is to replace the value of the left-hand side variable by the unevaluated expression on the right-hand side. Sometimes symbolic execution is combined with actual execution in order to simplify the terms being collected in variables. Most often, however, all variable

manipulations and decisions are made symbolically. As a consequence, all assignments become string assignments and all decision points are indeterminate. To illustrate a symbolic execution, consider the following small pseudocode program:

```
IN a, b;
a := a * a;
x := a + b;
IF x = 0 THEN x := 0
         ELSE x := 1;
```

The symbolic execution of the program will result in the following expression:

```
if a * a + b = 0 then x := 0
else if a * a + b ≠ 0 then x := 1
```

Note that we are unable to determine the result of the equality test for we only have symbolic values available.

The result of a symbolic execution is a large, complex expression that can be decomposed and viewed as a tree structure, where each leaf represents a path through the program. The symbolic values of each variable are known at every point within the tree and the branch points of the tree represent the decision points of the program. Every program path is represented in the tree, and every branch path is, by definition, taken.

If the program has no loops, then the resultant tree structure is finite, and can be used as an aid in generating test data that will cause every path in the program to be executed. The predicates at each branch point of the tree structure, for a particular path, are then collected into a single logical expression. Data that cause a particular path to be executed can be found by determining which data will make the path expression true. If the predicates are equalities, inequalities, and orderings, the problem of data selection becomes the classic problem of trying to solve a system of equalities and orderings. For more detail, see CLAR77 or HOWD77.

There are two major difficulties with using symbolic execution as a test set construction mechanism. The first is the combinatorial explosion inherent in the tree structure construction: the number of paths in the symbolic execution tree structure may grow as an exponential in the length of the program, leading to serious computational difficulties. If the program has loops, then the symbolic execution tree structure is necessarily infinite (since every predicate branch is taken). Usually only a finite number of loop executions is required, enabling a finite loop unwinding to be performed. The second difficulty is that the problem of determining whether the path expression has values that satisfy it is undecidable even with restricted programming languages [CHER79a]. For certain applications, however, symbolic execution has been successful in constructing test sets.

Another use of symbolic execution techniques is in the construction of verification conditions from partially annotated programs. Typically, the program has attached to each of its loops an assertion, called an "invariant," that is true at both the first and the last statement of the loop. (Thus the assertion remains "invariant" over one execution of the loop.) From this assertion, the programmer can construct an assertion that is true before entrance to the loop and an assertion that is true after exit of the loop. Such a program can then be viewed as free of loops (since each loop is considered as a single statement) and assertions can be extended to all statements of the program (so it is fully annotated) using techniques similar to those for symbolic execution. A good survey of these methods has been done by Hantler [HANT76], and an example of their use in verifiers appears in Luckham [LUCK79].

2.7.3 *Dynamic Analysis Techniques*

Dynamic analysis is usually a three-step procedure involving static analysis and instrumentation of a program, execution of the instrumented program, and finally, analysis of the instrumentation data. Often this is accomplished interactively through automated tools.

The simplest instrumentation technique for dynamic analysis is the insertion of a counter or "turnstile." Branch and segment coverage are determined in this manner. A preprocessor analyzes the program (usually by internally representing the program as

a program graph) and inserts counters at appropriate places.

For example, for IF statements, control will be directed, first, to a distinct statement responsible for incrementing a counter for each possible branch, and, second, back to the original statement. Two separate counters are employed when two IF statements branch to the same point. Loop constructs often have to be modified so that both interior and exterior paths can be instrumented. For example, the exterior path of a loop usually has no executable statements. To insert a counter, the loop construct must be modified, as below:

```
      DO 20 I = J, K, L
        .
        .
        .
 20   Statement k
      IF (I.GT.K) THEN 201
 20   N(20) = N(20) + 1
        .
        .
        .
      Statement k
      I = I + L
      IF (I.LE.K) THEN 20
201   N(201) = N(201) + 1
```

N(201) counts the exterior executions and N(20) counts the interior executions.

Simple statement coverage requires much less instrumentation than does either branch coverage or more extensive metrics. For complicated assignments and loop and branch predicates, more detailed instrumentation is employed. Besides simple counts, it is useful to know the maximum and minimum values of variables (particularly useful for array subscripts), the initial and final value, and other constraints particular to the application.

Instrumentation does not have to rely on direct code insertion. A simple alternate implementation is to insert calls to run-time routines in place of actual counters. The developer can insert commands in the code which is then passed through a preprocessor/compiler. The preprocessor adds the instrumentation only if the correct commands are set to enable it.

Stucki introduced the concept of instrumenting a program with *dynamic assertions*. A preprocessor generates instrumentation for dynamically checking conditions that are often as complicated as those used in program-proof techniques [STUC77]. These assertions are entered as comments in program code and are meant to be permanent. They provide both documentation and means for maintenance testing. All or individual assertions are enabled during test by using simple commands to the preprocessor.

There are assertions which can be employed globally, regionally, locally, or at entry and exit. The general form for a local assertion is

```
ASSERT LOCAL [optional qualifier]
    (extended-logical-expression) [control]
```

The optional qualifiers are adjectives such as ALL and SOME. The control options include (1) LEVEL, which controls the levels in a block-structured program; (2) CONDITIONS, which allows dynamic enabling of the instrumentation; and (3) LIMIT, which allows a specific number of violations to occur. The logical expression is used to represent an expected condition, which is then dynamically verified. For example, placing

```
ASSERT LOCAL
    (A(2 : 6, 2 : 10).NE.0) LIMIT 4
```

within a program will cause the values of array elements A(2, 2), A(2, 3),..., A(2, 10), A(3, 2),..., A(6, 10) to be checked against a zero value at each locality. After four violations during the execution of the program, the assertion will become false.

The global, regional, and entry–exit assertions are similar in structure to the local assertions described earlier. Note the similarity with proof-of-correctness techniques. These assertions are very much like the input, output, and intermediate assertions used in program proving (called verification conditions), especially if the entry–exit assertions are employed. Furthermore, symbolic execution can be used, just as it was with proof techniques, to generate the assertions. Some efforts are currently under way to integrate dynamic assertions, proof techniques, and symbolic evaluation. One of these is described below.

Andrews and Benson have described a system developed by General Research [ANDR81] that employs dynamic assertion

techniques in an automated test system. Code with embedded executable assertions can be tested using constrained optimization search strategies to vary an initial test data set over a range of test inputs, adapting the test data to the test results. The automated test system records the dynamic assertion evaluation for a large number of tests.

There are many other techniques for dynamic analysis. Most involve the dynamic (while under execution) measurement of the behavior of a part of a program, where the features of interest have been isolated and instrumented based on a static analysis. Some typical techniques include expression analysis, flow analysis, and timing analysis.

2.8 Combined Methods

There are many ways in which the techniques described above can be used in concert to form a more powerful and efficient testing technique. One of the more common combinations today merges standard testing techniques with formal verification. Our ability, through formal methods, to verify significant segments of code is improving [GERH78], and certain modules, either for security or reliability reasons, now justify the additional expense of formal verification.

Other possibilities for combination include using symbolic execution or formal proof techniques to verify those segments of code that, through coverage analysis, have been shown to be most frequently executed. Mutation analysis, for some special cases like decision tables, can be used to verify programs fully [BUDD78b]. Formal proof techniques may be useful in one of the problem areas of mutation analysis, the determination of equivalent mutants.

Another example, combining data-flow analysis, symbolic execution, elementary theorem proving, dynamic assertions, and standard testing is suggested by Osterweil [OSTE80]. Osterweil addresses the issue of how to combine efficiently these powerful techniques in one systematic method. As has been mentioned, symbolic evaluation can be used to generate dynamic assertions by first executing paths symbolically so that each decision point and every loop has an assertion, then checking for consistency using both data-flow and proof techniques. If all the assertions along a path are consistent, they can be reduced to a single dynamic assertion for the path. Either theorem-proving techniques can be used to "prove" the path assertion and termination, or dynamic testing methods can be used to test and evaluate the dynamic assertions for the test data.

Osterweil's technique allows for several trade-offs between testing and formal methods. For instance, symbolically derived dynamic assertions, although more reliable than manually derived assertions, cost more to generate. Consistency analysis of the assertions using proof and data-flow techniques adds cost to development, but reduces the number of repeated executions. Finally there is the overall trade-off between theorem proving and testing to verify the dynamic assertions.

3. CONCLUSIONS AND RESEARCH DIRECTIONS

We have surveyed many of the techniques used to validate software systems. Of the methods discussed, the most successful have been the disciplined manual techniques, such as walk-throughs, reviews, and inspections, applied to all stages in the life cycle [FAGA76]. Discovery of errors within the first stages of development (requirements and design) is particularly critical since the cost of these errors escalates significantly if they remain undiscovered until construction or later. Until the development products at the requirements and design stages become formalized, and hence amenable to automated analysis, disciplined manual techniques will continue to be the key verification techniques.

Many of the other techniques discussed in Section 2 have not seen wide use. These techniques appeal to our intuition, but we have only anecdotal evidence that they work. Howden showed in a study of a commercial FORTRAN-based scientific library [IMSL78, HOWD80b] that the success of particular testing technique does not correlate with structural or functional attributes of the code. It was this study that led

Howden to develop the ideas of design-based functional testing described in Section 2.4.

Recently Howden performed a similar study of a commercial COBOL-based general ledger system [HOWD81b], in which he found that the errors were much different from those in the IMSL library. As one might expect, errors in the data definition were much more common than errors in the procedures. Moreover, the most common errors were due to missing logic (i.e., various cases not being covered by program logic) and thus invisible to any structurally based technique. Glass [GLAS81] has noted similar experiences with embedded software. These experiences point up another problem that most of the techniques described in Section 2 are directed at procedural languages with only rudimentary input/output capability and are probably not as useful when applied to COBOL and similar languages. Test coverage will have to be more closely tied to the requirements to overcome this difficulty. Structural techniques based on data-flow coverage rather than control-flow coverage will need to be developed as well.

The Howden studies point to the major problem in testing: the lack of a sound theoretical foundation. Besides the work of Goodenough and Gerhart, Howden, and the Lipton, DeMillo, Sayward, and Budd mutation research we have made very little progress toward developing a theoretical basis from which to relate software behavior to validation and verification. While there have been efforts in this area by White [WHIT78], Clarke and Richardson [RICH81], Weyuker et al. [WEYU80, OSTR80, DAVI81], and others, it clearly requires considerably more research effort.

There are problems with these techniques other than just the lack of a sound theoretical basis. Many of the techniques have major costs associated with customizing them to the verification process (simulation) or high costs for their use (symbolic execution), or unproved applicability in practice (proof of correctness). Many of the techniques are areas of intense current research, but have not yet been developed or proven sufficiently in the real world. Only recently has validation and verification been given the attention it deserves in the development cycle. Budgets, except for a few highly critical software projects, have not included sufficient funds for adequate testing.

Even with these problems, the importance of performing validation throughout the life cycle is not diminished. One of the reasons for the great success of disciplined manual techniques is their uniform applicability at requirements, design, and coding phases. These techniques can be used without massive capital expenditure. However, to be most effective, they require a serious commitment and a disciplined application. Careful planning, clearly stated objectives, precisely defined techniques, good management, organized record keeping, and strong commitment are critical to successful validation.

We view the integration of validation with software development as crucial, and we suggest that it be an integral part of the requirements statement. Validation requirements should specify the type of manual techniques, the tools, the form of project management and control, the development methodology, and the acceptability criteria that are to be used during software development. These requirements are in addition to the functional requirements of the system ordinarily specified at this stage. If this practice were followed, embedded within the project requirements would be a statement of work aimed at enhancing the quality of the completed software.

A major difficulty with any proposal such as the above, however, is that we have neither the means of accurately measuring the effectiveness of validation methods nor the means of determining "how valid" the software should be. We assume that it is not possible to produce a "perfect" software system and take as our goal getting as close to perfect as can be reasonably (given these constraints) required. In addition, what constitutes perfect and how important it is for the software to be perfect may vary from project to project. Some software systems (such as those for reactor control) have more stringent quality requirements than other software (such as an address label program). Defining "perfect" (by specifying which quality attributes must be

met) and determining its importance should be part of the validation requirements. However, validation mechanisms written into the requirements do not guarantee "perfect" software, just as the use of a particular development methodology does not guarantee high-quality software. The evaluation of competing validation mechanisms will be difficult.

A further difficulty is that validation tools do not often exist in integrated packages. Since no one verification tool is sufficient, this means that the group performing the verification must acquire several tools and learn several methods that may be difficult to use in combination. This is a problem that must receive careful thought [ADRI80, BRAN81a], for, unless the combination is chosen judiciously, their use can lead to costs and errors beyond that necessary to acquire them in the first place. The merits of both the tool collection as a whole and of any single tool must be considered.

The efforts described in Section 2.9 to integrate verification techniques are very important. At present the key to high quality remains the disciplined use of a development methodology accompanied by verification at each stage of the development. No single technique provides a magic solution. For this reason, the integration of tools and techniques and the extension of these to the entire life cycle is necessary before adequate validation and verification becomes possible.

The current research on software support systems and programming environments [BRAN81b, BARS81a, BARS81b, WASS81a, WASS81b] can have major impact on validation and verification. The use of such environments has the potential to improve greatly the quality of the completed software. In addition, such systems may provide access by the user/customer to the whole process, providing a mechanism for establishing confidence in the quality of the software [CHER79b, CHER80].

Clearly, research is still necessary on the basic foundations of verification, on new tools and techniques, and on ways to integrate these into a comprehensive and automated development methodology. Moreover, given the increasing cost of software, both absolutely and as a proportion of total system cost, and the increasing need for reliability, it is important that management apply the needed resources and direction so that verification and validation can be effective.

4. GLOSSARY

Audit. See **DOD Development Reviews.**

Black Box Testing. See **Functional Testing.**

Boundary Value Analyses. A selection technique in which test data are chosen to lie along "boundaries" of input domain (or output range) classes, data structures, procedure parameters, etc. Choices often include maximum, minimum, and trivial values or parameters. This technique is often called stress testing. (See Section 2.4.)

Branch Testing. A test method satisfying coverage criteria that require that for each decision point each possible branch be executed at least once. (See Section 2.5.)

Cause-Effect Graphing. Test data selection technique. The input and output domains are partitioned into classes and analysis is performed to determine which input classes cause which effect. A minimal set of inputs is chosen that will cover the entire effect set. (See Section 2.4.)

Certification. Acceptance of software by an authorized agent usually after the software has been validated by the agent, or after its validity has been demonstrated to the agent.

Critical Design Review. See **DOD Development Reviews.**

Complete Test Set. A test set containing data that causes each element of a prespecified set of Boolean conditions to be true. Additionally, each element of the test set causes at least one condition to be true. (See Section 2.2.)

Consistent Condition Set. A set of Boolean conditions such that complete test sets for the conditions uncover the same errors. (See Section 2.2.)

Cyclomatic Complexity. The cyclomatic complexity of a program is equivalent to the number of decision statements plus 1. (See Section 2.5.)

DD (decision-to-decision) Path. A path of logical code sequence that begins at an entry or decision statement and ends at a decision statement or exit. (See Section 2.5.)

Debugging. The process of correcting syntactic and logical errors detected during coding. With the primary goal of obtaining an executing piece of code, debugging shares with testing certain techniques and strategies, but differs in its usual ad hoc application and local scope.

Design-Based Functional Testing. The application of test data derived through functional analysis (see **Functional Testing**) extended to include design functions as well as requirement functions. (See Section 2.4.)

DOD Development Reviews. A series of reviews required by DOD directives. These include

(1) The *Systems Requirements Review* is an examination of the initial progress during the problem definition stage and of the convergence on a complete system configuration. Test planning and test documentation are begun at this review.
(2) The *System Design Review* occurs when the system definition has reached a point where major system modules can be identified and completely specified along with the corresponding test requirements. The requirements for each major subsystem are examined along with the preliminary test plans. Tools required for verification support are identified and specified at this stage.
(3) The *Preliminary Design Review* is a formal technical review of the basic design approach for each major subsystem or module. The revised requirements and preliminary design specifications for each major subsystem and all test plans, procedures, and documentation are reviewed at this stage. Development and verification tools are further identified at this stage. Changes in requirements will lead to an examination of the test requirements to maintain consistency.
(4) The *Critical Design Review* occurs just prior to the beginning of the construction stage. The complete and detailed design specifications for each module and all draft test plans and documentation are examined. Again, consistency with previous stages is reviewed, with particular attention given to determining if test plans and documentation reflect changes in the design specifications at all levels.
(5) Two audits, the *Functional Configuration Audit* and the *Physical Configuration Audit* are performed. The former determines if the subsystem performance meets the requirements. The latter audit is an examination of the actual code. In both audits, detailed attention is given to the documentation, manuals and other supporting material.
(6) A *Formal Qualification Review* is performed to determine through testing that the final coded subsystem conforms with the final system specifications and requirements. It is essentially the subsystem acceptance test.

Driver. Code that sets up an environment and calls a module for test. (See Section 1.3.)

Dynamic Analysis. Analysis that is performed by executing the program code. (See Section 2.7.)

Dynamic Assertion. A dynamic analysis technique that inserts assertions about the relationship between program variables into the program code. The truth of the assertions is determined as the program executes. (See Section 2.7.)

Error Guessing. Test data selection technique. The selection criterion is to pick values that seem likely to cause errors. (See Section 2.4.)

Exhaustive Testing. Executing the program with all possible combinations of values for program variables. (See Section 2.1.)

Extremal Test Data. Test data that is at the extreme or boundary of the domain of an input variable or which produces results at the boundary of an output domain. (See Section 2.4.)

Formal Qualification Review. See **DOD Development Reviews.**

Functional Configuration Audit. See **DOD Development Reviews.**

Functional Testing. Application of test data derived from the specified functional requirements without regard to the final program structure. (See Section 2.4.)

Infeasible Path. A sequence of program statements that can never be executed. (See Section 2.5.)

Inspection. A manual analysis technique in which the program (requirements, design, or code) is examined in a very formal and disciplined manner to discover errors. (See Section 2.2.)

Instrumentation. The insertion of additional code into the program in order to collect information about program behavior during program execution. (See Section 2.7.)

Invalid Input (Test Data for Invalid Input Domain). Test data that lie outside the domain of the function the program represents. (See Section 2.1.)

Life-Cycle Testing. The process of verifying the consistency, completeness, and correctness of the software entity at each stage in the development. (See Section 1.)

Metric-Based Test Data Generation. The process of generating test sets for structural testing based upon use of complexity metrics or coverage metrics. (See Section 2.5.)

Mutation Analysis. A method to determine test set thoroughness by measuring the extent to which a test set can discriminate the program from slight variants (mutants) of the program. (See Section 2.6.)

Oracle. A mechanism to produce the "correct" responses to compare with the actual responses of the software under test. (See Section 2.1.)

Path Expressions. A sequence of edges from the program graph which represents a path through a program. (See Section 2.5.)

Path Testing. A test method satisfying coverage criteria that each logical path through the program be tested. Often paths through the program are grouped into a finite set of classes; one path from each class is then tested. (See Section 2.5.)

Preliminary Design Review. See **DOD Development Reviews.**

Program Graph. Graphical representation of a program. (See Section 2.5.)

Proof of Correctness. The use of techniques of mathematical logic to infer that a relation between program variables assumed true at program entry implies that another relation between program variables holds at program exit. (See Section 2.2.)

Regression Testing. Testing of a previously verified program required following program modification for extension or correction. (See Section 1.4.)

Simulation. Use of an executable model to represent the behavior of an object. During testing the computational hardware, the external environment, and even code segments may be simulated. (See Section 2.2.)

Self-Validating Code. Code which makes an explicit attempt to determine its own correctness and to proceed accordingly. (See Section 2.7.)

Special Test Data. Test data based on input values that are likely to require special handling by the program. (See Section 2.4.)

Statement Testing. A test method satisfying the coverage criterion that each statement in a program be executed at least once during program testing. (See Section 2.5.)

Static Analysis. Analysis of an program that is performed without executing the program. (See Section 2.7.)

Stress Testing. See **Boundary Value Analysis.**

Structural Testing. A testing method where the test data are derived solely from the program structure. (See Section 2.5.)

Stub. Special code segments that, when invoked by a code segment under test, will simulate the behavior of designed and specified modules not yet constructed. (See Section 1.3.)

Symbolic Execution. A static analysis technique that derives a symbolic expression for each program path. (See Section 2.7.)

System Design Review. See **DOD Development Reviews.**

System Requirements Review. See **DOD Development Reviews.**

Test Data Set. Set of input elements used in the testing process. (See Section 2.1.)

Test Driver. A program that directs the execution of another program against a collection of test data sets. Usually the test driver also records and organizes the output generated as the tests are run. (See Section 1.3.)

Test Harness. See **Test Driver.**

Testing. Examination of the behavior of a program by executing the program on sample data sets.

Valid Input (test data for a valid input domain). Test data that lie within the domain of the function represented by the program. (See Section 2.1.)

Validation. Determination of the correctness of the final program or software produced from a development project with respect to the user needs and requirements. Validation is usually accomplished by verifying each stage of the software development life cycle.

Verification. In general, the demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the development life cycle.

Walk-Through. A manual analysis technique in which the module author describes the module's structure and logic to an audience of colleagues. (See Section 2.2.)

White Box Testing. See **Structural Testing.**

REFERENCES

ACRE80 ACREE, A. "On Mutation," Ph.D dissertation, Information and Computer Science Dep, Georgia Institute of Technology, Atlanta, June, 1980

ADRI80 ADRION, W R. "Issues in software validation, verification, and testing," *ORSA/TIMS Bull.* (1980 TIMS-ORSA Conf.) 10 (Sept. 1980), 80.

ALFO77 ALFORD, M W "A requirement engineering methodology for real-time processing requirements," *IEEE Trans Softw Eng* **SE-2,** 1 (1977), 60-69.

ALLE74 ALLEN, F. E. "Interprocedural data flow analysis," in *Proc IFIP Congress 1974,* North-Holland, Amsterdam, 1974, pp. 398-402.

ALLE76 ALLEN, F. E, AND COCKE, J. "A program data flow procedure," *Commun. ACM* **19,** 3 (March 1976), 137-147.

AMBL78 AMBLER, A. L., GOOD, D. I., BROWNE, J C, BURGER, W F, COHEN, R. M, HOCH, C. G, AND WELLS, R. E "Gypsy: A language for specification and implementation of verifiable programs," in *Proc. Conf. Language Design for Reliable Software,* D. B. Wortman (Ed.), ACM, New York, pp. 1-10.

ANDR81 ANDREWS, D. M., AND BENSON, J. P. "An automated program testing methodology, and its implementation," in *Proc. 5th Int. Conf. Software Engineering* (San Diego, Calif., March 9-12), IEEE Computer Society Press, Silver Spring, Maryland, 1981, pp. 254-261.

ANSI78 ANSI X3 9-1978, "FORTRAN," American National Standards Institute, New York, 1978.

APT81 APT, K. R., "Ten years of Hoare's logic: A survey—Part I," *Trans. Program Lang. Syst.* **3,** 4 (Oct. 1981), 431-483.

BAKE72 BAKER, F. T. "Chief programmer team management of production programming," *IBM Syst. J* **11,** 1 (1972), 56-73.

BARS81a BARSTOW, D. R., AND SHROBE, H. E. (Eds). Special Issue on Programming Environments, *IEEE Trans. Softw. Eng.* **SE-7,** 5 (Sept. 1981).

BARS81b BARSTOW, D R., SHROBE, H, AND SANDEWALL, E., Eds. *Interactive programming environments,* McGraw-Hill, New York, 1981

BELL77 BELL, T. E., BIXLER, D. C., AND DYER, M. E. "An extendable approach to computer-aided software requirements engineering," *IEEE Trans. Softw Eng.* **SE-3,** 1 (1977), 49-60.

BOEH77 BOEHM, B. W., "Seven basic principles of software engineering," in *Software engineering techniques,* Infotech State of the Art Report, Infotech, London, 1977.

BOEH78 BOEHM, B. W, BROWN, J. R., KASPAR, H., LIPOW, M., MACLEOD, G. J, AND MERRIT, M. J. *Characteristics of software quality,* North-Holland, New York, 1978.

BOYE75 BOYER, R. S., ELSPAS, B., AND LEVITT, K. N. "SELECT—A formal system for testing and debugging programs by symbolic execution," in *Proc. 1975 Int. Conf. Reliable Software* (Los Angeles, April), 1975, pp 234-245.

BRAN80 BRANSTAD, M. A, CHERNIAVSKY, J. C., AND ADRION, W. R "Validation, verification, and testing for the individual

programmer," *Computer* **13,** 12 (Dec 1980), 24-30

BRAN81a BRANSTAD, M. A., ADRION, W. R., AND CHERNIAVSKY, J. C. "A view of software development support systems," in *Proc. Nat Electronics Conf.*, vol. 35, National Engineering Consortium, Oakbrook, Ill., Oct. 1981, pp. 257-262.

BRAN81b BRANSTAD, M. A , AND ADRION, W. R., Eds. "NBS programming environment workshop," *Softw. Eng Notes* **6,** 4 (Aug 1981), 1-51.

BROW73 BROWN, J.R., ET AL. "Automated software quality assurance," in W Hetzel (Ed.), *Program test methods*, Prentice-Hall, Englewood Cliffs, N.J , 1973, Chap 15.

BUCK79 BUCKLEY, F. "A standard for software quality assurance plans," *Computer* **12,** 8 (Aug 1979), 43-50

BUDD78a BUDD, T , DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. "The design of a prototype mutation system for program testing," in *Proc AFIPS Nat Computer Conf.*, vol 47, AFIPS Press, Arlington, Va., 1978, pp 623-627.

BUDD78b BUDD, T A., AND LIPTON, R. J. "Mutation analysis of decision table programs," in *Proc. 1978 Conf. Information Science and Systems*, Johns Hopkins Univ , Baltimore, Md., pp. 346-349.

CAIN75 CAINE, S. H , AND GORDON, E. K. "PDL—Baltimore, A tool for software design," in *Proc National Computer Conf.*, vol. 44, AFIPS Press, Arlington, Va., 1975, pp. 271-276.

CARP75 CARPENTER, L. C., AND TRIPP, L. L. "Software design validation tool," in *Proc 1975 Int Conf Reliable Software* (Apr 1975)

CHAP79 CHAPIN, N "A measure of software complexity," in *Proc. AFIPS National Computer Conf.*, vol 48, AFIPS Press, Arlington, Va., 1979, pp 995-1002.

CHER79a CHERNIAVSKY, J. C "On finding test data sets for loop free programs," *Inform. Process. Lett* **8,** 2 (1979).

CHER79b CHERNIAVSKY, J. C., ADRION, W R., AND BRANSTAD, M. A. "The role of testing tools and techniques in the procurement of quality software and systems," in *Proc. 13th Annu. Alisomar Conf Circuits, Systems, and Computers*, IEEE Computer Society, Long Beach, Calif., 1979, pp. 309-313

CHER80 CHERNIAVSKY, J C., ADRION, W. R., AND BRANSTAD, M. A. "The role of programming environments in software quality assurance," in *Proc. Nat Electronics Conf.*, vol. 34, National Engineering Consortium, Oakbrook, Ill , 1980, pp. 468-472

CLAR77 CLARKE, A. "A system to generate test data and symbolically execute programs," *IEEE Trans Softw Eng.* **SE-2,** 3 (Sept. 1977), 215-222.

CONS78 CONSTABLE, R. L., AND O'DONNELL, M. J. *A programming logic,* Winthrop, Cambridge, Mass., 1978.

DAVI81 DAVIS, M. D., AND WEYUKER, E. J "Pseudo-oracles for montestable programs," Tech. Rep., Courant Institute of Mathematical Sciences, New York, 1981.

DEMI78 DEMILLO, R. A., LIPTON, R. J , AND SAYWARD, F.G. "Hints on test data selection: Help for the practicing programmer," *Computer* **11,** 4 (1978), 34-43.

DEMI79 DEMILLO, R. A., LIPTON, R. J., AND PERLIS, A. J. "Social processes and the proofs of theorems and programs," *Commun. ACM* **225** (May 1979), 271-280.

DIJK72 DIJKSTRA, E.W. "Notes on structured programming," in O. J. Dahl, E. J. Dijkstra, and C. A. R. Hoare (Eds.), *Structured programming*, Academic Press, London, 1972.

DIJK78 DIJKSTRA, E. W. "On a political pamphlet from the Middle Ages (regarding the POPL paper of R. A. DeMillo, R. J. Lipton, and A. J. Perlis)," *Softw Eng Notes* **3,** 2 (Apr. 1978), 14-15.

DURA81a DURAN, J. W., AND WIORKOWSKI, J J "Capture-recapture sampling for estimating software error content," *IEEE Trans Softw. Eng* **SE-7** (Jan 1981), 147-148.

DURA81b DURAN, J. W., AND NTAFOS, S. "A report on random testing," in *Proc. 5th Int. Conf. Software Engineering*, IEEE Computer Society Press, Silver Spring, Md , 1981, pp. 179-183.

EDP81 *EDP Analyzer,* vol. 9, 8 (Aug. 1981).

FAGA76 FAGAN, M. E. "Design and code inspections to reduce errors in program development," *IBM Syst. J* **15,** 3 (1976), 182-211.

FIPS76 FIPS. "Guidelines for documentation of Computer Programs and Automated Data Systems," FIPS38, Federal Information Processing Standards Publications, U.S. Department of Commerce/National Bureau of Standards, Washington, D.C., 1976.

FLOY67 FLOYD, R. W. "Assigning meaning to programs," in *Proc. Symposia Applied Mathematics*, vol. 19, American Mathematics Society, Providence, R.I., 1967, pp. 19-32.

FOSD76 FOSDICK, L. D., AND OSTERWEIL, L. J "Data flow analysis in software reliability," *Comput Surv* (ACM) **8,** 3 (Sept. 1976), 305-330.

GAO81 GENERAL ACCOUNTING OFFICE "Federal agencies' maintenance of computer programs· Expensive and undermanaged," GAO, Washington, D C , 1981

GERH78 GERHART, S. L. "Program verification in the 1980s: Problems, perspectives, and opportunities," Rep. ISI/RR-78-71,

Information Sciences Institute, Marina del Rey, Calif., Aug. 1978.

GERH80 GERHART, S. L., MUSSER, D. R., THOMPSON, D. H., BAKER, D. A., BATES, R. L., ERICKSON, R. W., LONDON, R. L., TAYLOR, D. G., AND WILE, D. S. "An overview of AFFIRM, A specification and verification system," in *Proc. IFIP Congress 1980*, North-Holland, Amsterdam, pp. 343–347

GLAS81 GLASS, R. L. "Persistent software errors," *IEEE Trans. Softw. Eng.* **SE-7,** 2 (March 1981), 162–168.

GOOD75 GOODENOUGH, J. B., AND GERHART, S. L. "Toward a theory of test data selection," *IEEE Trans. Softw. Eng* **SE-1,** 2 (March 1975).

HALS77 HALSTEAD, M. H. *Elements of software science,* Elsevier North-Holland, New York, 1977

HAMI76 HAMILTON, N., AND ZELDIN, S. "Higher order software—A methodology for defining software," *IEEE Trans. Softw. Eng* **SE-2,** 1 (1976), 9–32.

HANT76 HANTLER, S. L., AND KING, J. C. "An introduction to proving the correctness of programs," *Comput. Surv.* (ACM) **8,** 3 (Sept. 1976), 331–353

HECH72 HECHT, M., AND ULLMAN, J. "Flowgraph reducibility," *SIAM J. Appl. Math* **1** (1972), 188–202.

HOWD76 HOWDEN, W. E. "Reliability of the path analysis testing strategy," *IEEE Trans. Softw. Eng.* **SE-2,** 3 (1976).

HOWD77 HOWDEN, W. E. "Symbolic testing and the DISSECT symbolic evaluation system," *IEEE Trans. Softw. Eng* **SE-3,** 4 (1977), 266–278

HOWD78 HOWDEN, W. E. "A survey of dynamic analysis methods," in E. Miller and W. E. Howden (Eds.), *Tutorial. Software testing and validation techniques,* IEEE Computer Soc., New York, 1978

HOWD80a HOWDEN, W. E. "Functional program testing," *IEEE Trans. Soft. Eng* **SE-6,** 2 (1980), 162–169

HOWD80b HOWDEN, W. E. "Applicability of software validation techniques to scientific programs," *Trans. Program. Lang. Syst* **2,** 3 (June 1980), 307–320.

HOWD81a HOWDEN, W. E. "Completeness criteria for testing elementary program functions," in *Proc. 5th Int. Conf. on Software Engineering* (San Diego, March 9–12), IEEE Computer Society Press, Silver Spring, Md., 1981, pp. 235–243.

HOWD81b HOWDEN, W. E. "Errors in data processing programs and the refinement of current program test methodologies," Final Rep., NBS Contract NB79BCA0069, National Bureau of Standards, Washington, D.C., July 1981.

IEEE79 IEEE. Draft Test Documentation Standard, IEEE Computer Society Technical Committee on Software Engineering, Subcommittee on Software Standards, New York, 1979

IMSL78 IMSL. *Library reference manual.* International Mathematical and Statistical Libraries, Houston, Tex., 1978.

INFO79 INFOTECH *Software testing, INFOTECH state of the art report,* Infotech, London, 1979

JACK79 JACKSON, M. A. *Principles of program design,* Academic Press, New York, 1975.

JONE76 JONES, C. "Program quality and programmer productivity," IBM Tech. Rep., International Business Machines Corp., San Jose, Calif., 1976.

KERN74 KERNIGHAN, B. W. "RATFOR—A preprocessor for a rational FORTRAN," Bell Labs. Internal Memorandum, Bell Laboratories, Murray Hill, N.J., 1974.

KING76 KING, J. C. "Symbolic execution and program testing," *Commun. ACM* **19,** 7 (July 1976), 385–394.

KOPP76 KOPPANG, R. G. "Process design system—An integrated set of software development tools," in *Proc. 2nd Int. Software Engineering Conf* (San Francisco, Oct. 13–15), IEEE, New York, 1976, pp. 86–90.

LAMB78 LAMB, S. S., LECK, V. G., PETERS, L. J., AND SMITH, G. L. "SAMM: A modeling tool for requirements and design specification," in *Proc. COMPSAC 78,* IEEE Computer Society, New York, 1978, pp. 48–53.

LIPT78 LIPTON, R. J., AND SAYWARD, F. G. "The status of research on program mutation," in *Proc. Workshop on Software Testing and Test Documentation,* IEEE Computer Society, New York, 1978, pp. 355–367.

LUCK79 LUCKHAM, D., GERMAN, S., VON HENKE, F., KARP, R., MILNE, P., OPPEN, D., POLAK, W., AND SCHENLIS, W. "Stanford Pascal Verifier user's manual," AI Memo. CS-79-731, Computer Science Dep., Stanford University, Stanford, Calif., 1979.

LYON74 LYON, G., AND STILLMAN, R.B. "A FORTRAN analyzer," NBS Tech. Note 849, National Bureau of Standards, Washington, D.C., 1974.

MAIT80 MAITLAND, R. "NODAL," in *NBS software tools database,* R. Houghton and K. Oakley (Eds.), NBSIR, National Bureau of Standards, Washington, D.C., 1980.

MANN74 MANNA, Z. *Mathematical theory of computation,* McGraw-Hill, New York, 1974

MCCA76 MCCABE, T. J. "A complexity measure," *IEEE Trans. Softw. Eng* **SE-2,** 4 (1976), 308–320

McCa77 McCall, J., Richards, P., and Walters, G *Factors in software quality,* vols. 1-3, NTIS Rep File Nos. AD-A049-014, 015, 055, 1977.

Metz77 Metzner, J. R., and Barnes, B. H *Decision table languages and systems,* Academic Press, New York, 1977.

Mill70 Mills, H D "Top down programming in large systems," in *Debugging techniques in large systems,* R. Rustin (Ed), Prentice-Hall, Englewood Cliffs, N J., 1970, pp 41-55.

Mill72 Mills, H. D "On statistical validation of computer programs," IBM Rep. FSC72-6015, Federal Systems Division, IBM, Gaithersburg, Md , 1972.

Mill75 Miller, E F., Jr. "RXVP—An automated verification system for FORTRAN," in *Proc. Workshop 4, Computer Science and Statistics· 8th Ann. Symp on the Interface* (Los Angeles, Calif , Feb), 1975.

Mill77 Miller, E. R., Jr. "Program testing Art meets theory," *Computer* **10,** 7 (1977), 42-51

MilS76 Military Standard. "Technical reviews and audits for systems, equipment, and computer programs," MIL-STD-1521A (USAF), U.S. Department of the Air Force, Washington, D.C , 1976.

Myer76 Myers, G. J. *Software reliability—Principles and practices,* Wiley, New York, 1976

Myer79 Myers, G. J. *The art of software testing,* Wiley, New York, 1979.

Neum75 Neumann, P G., Robinson, L , Levitt, K., Boyer, R S., and Saxema, A. R. "A provably secure operating system," SRI Project 2581, SRI International, Menlo Park, Calif., 1975

Oste76 Osterweil, L. J., and Fosdick, L. D. "DAVE—A validation, error detection, and documentation system for FORTRAN programs," *Softw. Pract. Exper* **6** (1976), 473-486

Oste80 Osterweil, L. J. "A strategy for effective integration of verification and testing techniques," Tech Rep. CU-CS-181-80, Computer Science Dep., Univ. of Colorado, Boulder, 1980.

Ostr80 Ostrand, T. J., and Weyucker, E J "Current directions in the theory of testing," in *Proc IEEE Computer Software and Applictions Conf. (COMPSAC80),* IEEE Press, Silver Spring, Md , 1980, pp. 386-389.

Paig77 Paige, M. R. "On partitioning program graphs," *IEEE Trans. Softw Eng.* **SE-3,** 6 (1977), 87, 386-393

Panz78 Panzl, D J "Automatic revision of formal test procedures," in *Proc 3rd Int Conf Software Engineering* (Atlanta, May 10-12), ACM, New York, 1978, pp. 320-326

Parn77 Parnas, D L "The use of precise specifications in the development of software," in *Information processing 77,* B. Gilchrist (Ed.), North-Holland, Amsterdam, 1977, pp. 861-867.

Prat77 Pratt, V R. "Semantic considerations in Floyd-Hoare logic," in *Proc. 17th Annu. IEEE Symp. on the Foundations of Computer Science,* IEEE Computer Society Press, Long Beach, Calif , 1976, pp. 109-112.

Rama74 Ramamoorthy, C. V., and Ho, S F. FORTRAN automated code evaluation system, ERL—M466, Electronics Research Lab., Univ. of California, Berkeley, 1974.

Rich81 Richardson, D. J., and Clarke, L. A. "A partition analysis method to increase program reliability," in *Proc 5th Int Conference SoftwareEngineering* (San Diego, March 9-12), IEEE Computer Society Press, Silver Spring, Md , 1981, pp. 244-253.

Robi79 Robinson, L. *The HDM handbook,* vol. I-III, SRI Project 4828, SRI International, Menlo Park, Calif., 1979.

Ross77 Ross, D T., and Schoman, K. E., Jr. "Structured analysis for requirements definition," *IEEE Trans Softw. Eng.* **SE-3,** 1 (1977), 6-15.

Roub76 Roubine, O., and Robinson, L. Special Reference Manual, Stanford Research Institute Tech. Rep CSG-45, Menlo Park, Calif., 1976.

Schi78 Schick, G. J., and Wolverton, R. W. "An analysis of competing software reliability models," *IEEE Trans. Softw Eng.* **SE-4** (March, 1978), 104-120.

Snee78 Sneed, H., and Kirchoff, K "Prufstand—A testbed for systematic software components," in *Proc INFOTECH State of the Art Conf. Software Testing,* Infotech, London, 1978.

SRS79 SRS *Proc. Specifications of Reliable Software Conference,* IEEE Catalog No. CH1401-9C, IEEE, New York, 1979.

Stuc77 Stucki, L. G. "New directions in automated tools for improving software quality," in R. Yeh (Ed), *Current trends in programming methodology,* vol II—*Program validation,* Prentice-Hall, Englewood Cliffs, N J., 1977, pp. 80-111.

Taus77 Tausworthe, R. C. *Standardized development of computer software,* Prentice-Hall, Englewood Cliffs, N J., 1977.

Teic77 Teichroew, D., and Hershey, E. A , III "PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems," *IEEE Trans. Softw. Eng* **SE-3,** 1 (Jan. 1977), 41-48.

Wass81a Wasserman, A. (Ed.). Special Issue

on Programming Environments, *Computer* **14,** 4 (Apr. 1981).

WASS81b WASSERMAN, A. (Ed.). *Tutorial: Software development environments,* IEEE Computer Society, Silver Spring, Md., 1981.

WEIN71 WEINBERG, G M *The psychology of computer programming,* Van Nostrand-Reinhold, Princeton, N J., 1971

WEYU80 WEYUCKER, E. J., AND OSTRAND, T. J. "Theories of program testing and the application of revealing subdomains," *IEEE Trans. Softw. Eng.* **SE-6** (May, 1980), 236-246.

WHIT78 WHITE, L. J., AND COHEN, E. I. "A domain strategy for computer program testing," *Digest for the Workshop on Software Testing and Test Documentation* (Ft. Lauderdale, Fla), pp 335-354. Also appears in *IEEE Trans. Softw. Eng.* **SE-6** (May 1980), 247-257.

YOUR79 YOURDON, E., AND CONSTANTINE, L. L. *Structured design,* Prentice-Hall, Englewood Cliffs, N.J., 1979.

ZELK78 ZELKOWITZ, M. V. "Perspectives on software engineering," *Comput. Surv.* (ACM) **10,** 2 (June 1978), 197-216

Received January 1980; final revision accepted March 1982