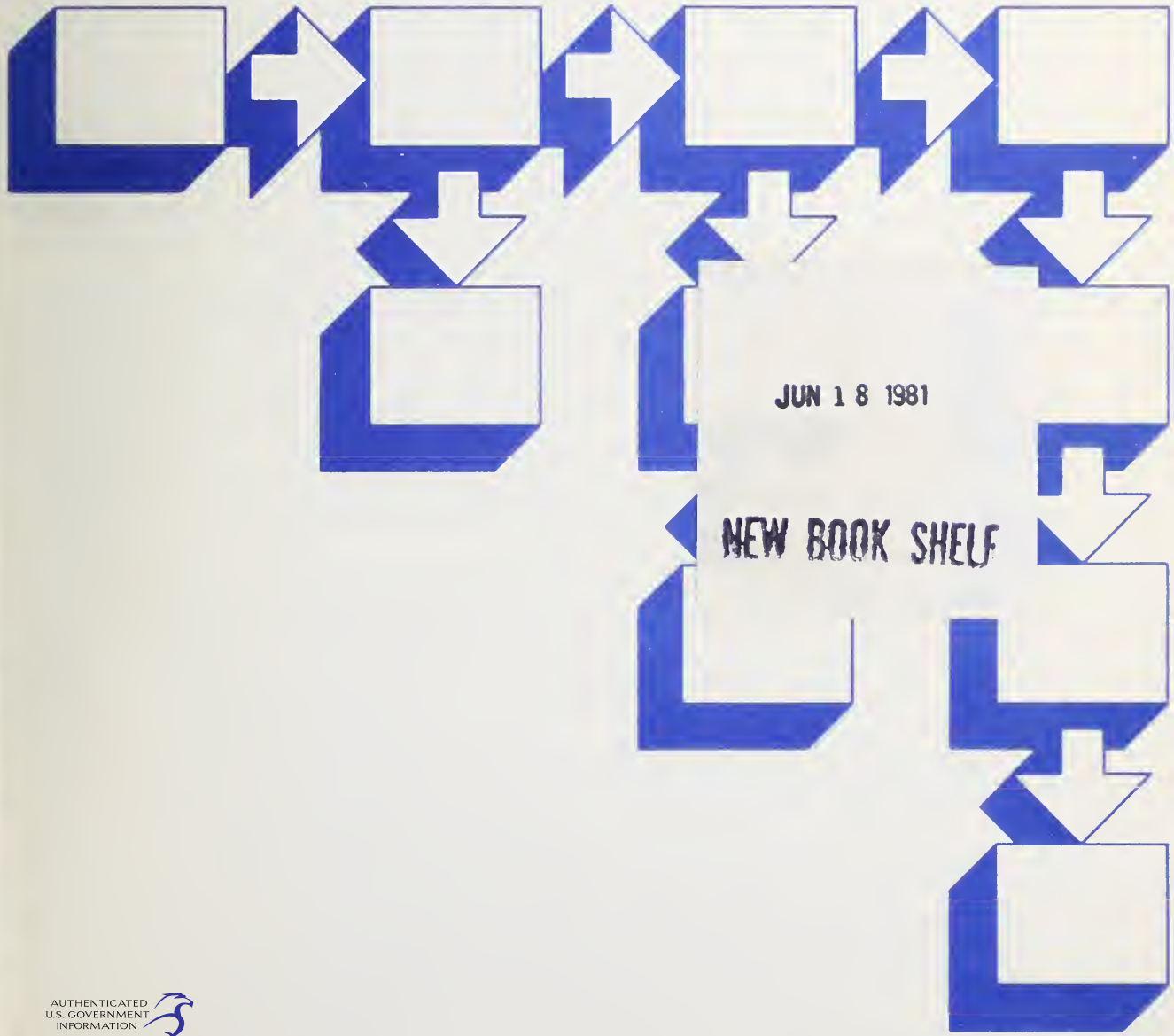
U.S. Department
of Commerce

National Bureau
of Standards

Computer Science and Technology

NBS Special Publication 500-75

Validation, Verification, and Testing of Computer Software



NATIONAL BUREAU OF STANDARDS

The National Bureau of Standards[1] was established by an act of Congress on March 3, 1901. The Bureau's overall goal is to strengthen and advance the Nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research and provides: (1) a basis for the Nation's physical measurement system, (2) scientific and technological services for industry and government, (3) a technical basis for equity in trade, and (4) technical services to promote public safety. The Bureau's technical work is performed by the National Measurement Laboratory, the National Engineering Laboratory, and the Institute for Computer Sciences and Technology.

THE NATIONAL MEASUREMENT LABORATORY provides the national system of physical and chemical and materials measurement; coordinates the system with measurement systems of other nations and furnishes essential services leading to accurate and uniform physical and chemical measurement throughout the Nation's scientific community, industry, and commerce; conducts materials research leading to improved methods of measurement, standards, and data on the properties of materials needed by industry, commerce, educational institutions, and Government; provides advisory and research services to other Government agencies; develops, produces, and distributes Standard Reference Materials; and provides calibration services. The Laboratory consists of the following centers:

Absolute Physical Quantities[2] — Radiation Research — Thermodynamics and Molecular Science — Analytical Chemistry — Materials Science.

THE NATIONAL ENGINEERING LABORATORY provides technology and technical services to the public and private sectors to address national needs and to solve national problems; conducts research in engineering and applied science in support of these efforts; builds and maintains competence in the necessary disciplines required to carry out this research and technical service; develops engineering data and measurement capabilities; provides engineering measurement traceability services; develops test methods and proposes engineering standards and code changes; develops and proposes new engineering practices; and develops and improves mechanisms to transfer results of its research to the ultimate user. The Laboratory consists of the following centers:

Applied Mathematics — Electronics and Electrical Engineering[2] — Mechanical Engineering and Process Technology[2] — Building Technology — Fire Research — Consumer Product Technology — Field Methods.

THE INSTITUTE FOR COMPUTER SCIENCES AND TECHNOLOGY conducts research and provides scientific and technical services to aid Federal agencies in the selection, acquisition, application, and use of computer technology to improve effectiveness and economy in Government operations in accordance with Public Law 89-306 (40 U.S.C. 759), relevant Executive Orders, and other directives; carries out this mission by managing the Federal Information Processing Standards Program, developing Federal ADP standards guidelines, and managing Federal participation in ADP voluntary standardization activities; provides scientific and technological advisory services and assistance to Federal agencies; and provides the technical foundation for computer-related policies of the Federal Government. The Institute consists of the following centers:

Programming Science and Technology — Computer Systems Engineering.

[1]Headquarters and Laboratories at Gaithersburg, MD, unless otherwise noted; mailing address Washington, DC 20234.
[2]Some divisions within the center are located at Boulder, CO 80303.

Computer Science and Technology

NBS Special Publication 500-75

Validation, Verification, and Testing of Computer Software

W. Richards Adrion
Martha A. Branstad
John C. Cherniavsky

Center for Programming Science and Technology
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, DC 20234

U.S. DEPARTMENT OF COMMERCE
Philip M. Klutznick, Secretary

Jordan J. Baruch, Assistant Secretary for Productivity, Technology and Innovation

National Bureau of Standards
Ernest Ambler, Director

Issued February 1981

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

National Bureau of Standards Special Publication 500-75
Nat. Bur. Stand. (U.S.), Spec. Publ. 500-75, 62 pages (Feb. 1981)
CODEN: XNBSAV

Library of Congress Catalog Card Number: 80-600199

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1980

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402
Price $3.75
(Add 25 percent for other than U.S. mailing)

TABLE OF CONTENTS

Page

Chapter 1: Quality Assurance Through Verification 2

1.1 Attributes of Quality Software 2

1.2 Verification Throughout the Lifecycle 6

REQUIREMENTS 8
DESIGN 8
CONSTRUCTION 9
MAINTENANCE 10

Chapter 2 An Overview of Testing 11

2.1 Concepts 11

2.2 Dynamic Testing 12

2.3 Structural vs Functional Testing 13

2.4 Static Testing 14

2.5 Manual vs Automated Testing 14

Chapter 3 Verification Techniques 16

3.1 General Verification Techniques 16

DESK CHECKING AND PEER REVIEW 16
WALKTHROUGHS, INSPECTIONS, AND REVIEWS 17
PROOF OF CORRECTNESS TECHNIQUES 19
SIMULATION 20

3.2 Test Data Generation 22

3.3 Functional Testing Techniques 23

BOUNDARY VALUE ANALYSIS 23
ERROR GUESSING AND SPECIAL VALUE ANALYSIS 24
CAUSE EFFECT GRAPHING 25
DESIGN BASED FUNCTIONAL TESTING 25

3.4 Structural Testing Techniques 26

COVERAGE-BASED TESTING 26
COMPLEXITY-BASED TESTING 28

3.5 Test Data Analysis 30

STATISTICAL ANALYSES AND ERROR SEEDING 30
MUTATION ANALYSIS 32

3.6 Static Analysis Techniques 32

FLOW ANALYSIS 33
SYMBOLIC EXECUTION 35

3.7 Dynamic Analysis Techniques 36

3.8 Combined Methods 39

3.9 Test Support Tools 39

TEST DOCUMENTATION 40
TEST DRIVERS 40
AUTOMATIC TEST SYSTEMS AND TEST LANGUAGES 41

Chapter 4 Summary 43

GLOSSARY 46

REFERENCES 51

VALIDATION, VERIFICATION, AND TESTING OF COMPUTER SOFTWARE

W. Richards Adrion
Martha A. Branstad
John C. Cherniavsky

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 programming product. Verification throughout the development process is stressed. Specific tools and techniques are described.

Key words: Validation; software verification; software testing; test data generation; test coverage; automated software tools; software lifecycle.

The National Bureau of Standards (NBS) has a mission under Public Law 89-306 (Brooks Act) to develop standards to enable the "economic and efficient purchase, lease, maintenance, operation, and utilization of automatic data processing equipment by Federal Departments and agencies." As part of its current standards initiative, NBS is studying methods to ensure the quality of software procured by the Government and software developed within the Government.

Testing is the traditional technique used to determine and assure the quality of products. For many items procured by the Government, the definition or description of a quality product and the testing methods used to ensure that quality are well established. These tests are usually physical tests based on both industry and Government standards (such as dimensions for fittings, strength for materials, power for motors, etc.). The success of these methods depends upon the definition of what constitutes a quality product, the determination of measurable properties that reflect the quality, the derivation of meaningful test criteria based on the measurable quantities, and the formulation of adequate tests to ensure the quality.

Unfortunately, software does not fit into the traditional framework of quality assessment. One reason is that software, in general, is a "one of a kind" product especially tailored for a particular application. There is often no standard product or specification to use as a model to measure against. Secondly, analogies to physical products with applicable dimensional, strength, etc. standards do not exist. Of greatest importance, the concept of what constitutes quality in software is not as well formulated. There is no universally accepted definition of software quality.

1.1 Attributes of Quality Software

There have been many studies directed toward the determination of appropriate factors for software quality [BOEH78], [MCCA77], [JONE76]. A number of attributes have been proposed; the set given by Figure 1.1 is representative. Most of these factors are qualitative rather than quantitative.

In Figure 1.1, the top level characteristics of quality software are reliability, testability, usability, efficiency, transportability, and maintainability. In practice, efficiency often turns out to be in conflict with other attributes, e.g. transportability, maintainability, and testability. As hardware costs decrease, efficiency of machine use becomes much less an

issue and consequently a less important attribute of software quality. At present, a reasonable software development methodology will support the creation of software with all these qualities. While a piece of code may not be locally as efficient as a skilled programmer can write it disregarding all other factors, it must be designed to be as efficient as possible while still exhibiting the other desired qualities.

For the purpose of this document, two qualities stand out, reliability and testability. The others are equally important, but less related to testing and verification issues, and perhaps more qualitative than quantitative. Reliable software must be adequate, that is, it must be correct, complete, consistent, and feasible at each stage of the development lifecycle. An infeasible set of requirements will lead to an inadequate design and probably an incorrect implementation. Given that the software meets these adequacy requirements at each stage of the development process, to be reliable it must also be robust. Robustness is a quality which represents the ability of the software to survive a hostile environment. We cannot anticipate all possible events [ADRI8Ø], and we must build our software to be as resilient as possible.

At all stages of the lifecycle, software should be testable. To accomplish this it must be understandable. The desired product (the requirements and design) and the actual product (the code) should be represented in a structured, concise, and self-descriptive manner so that they can be compared. The software must also be measurable, allowing means for actually instrumenting or inserting probes, testing, and evaluating the product of each stage.

Emphasis on particular quality factors will vary from project to project depending on application, environment, and other considerations. The specific definition of quality and the importance of given attributes should be specified during the requirements phase of the project.

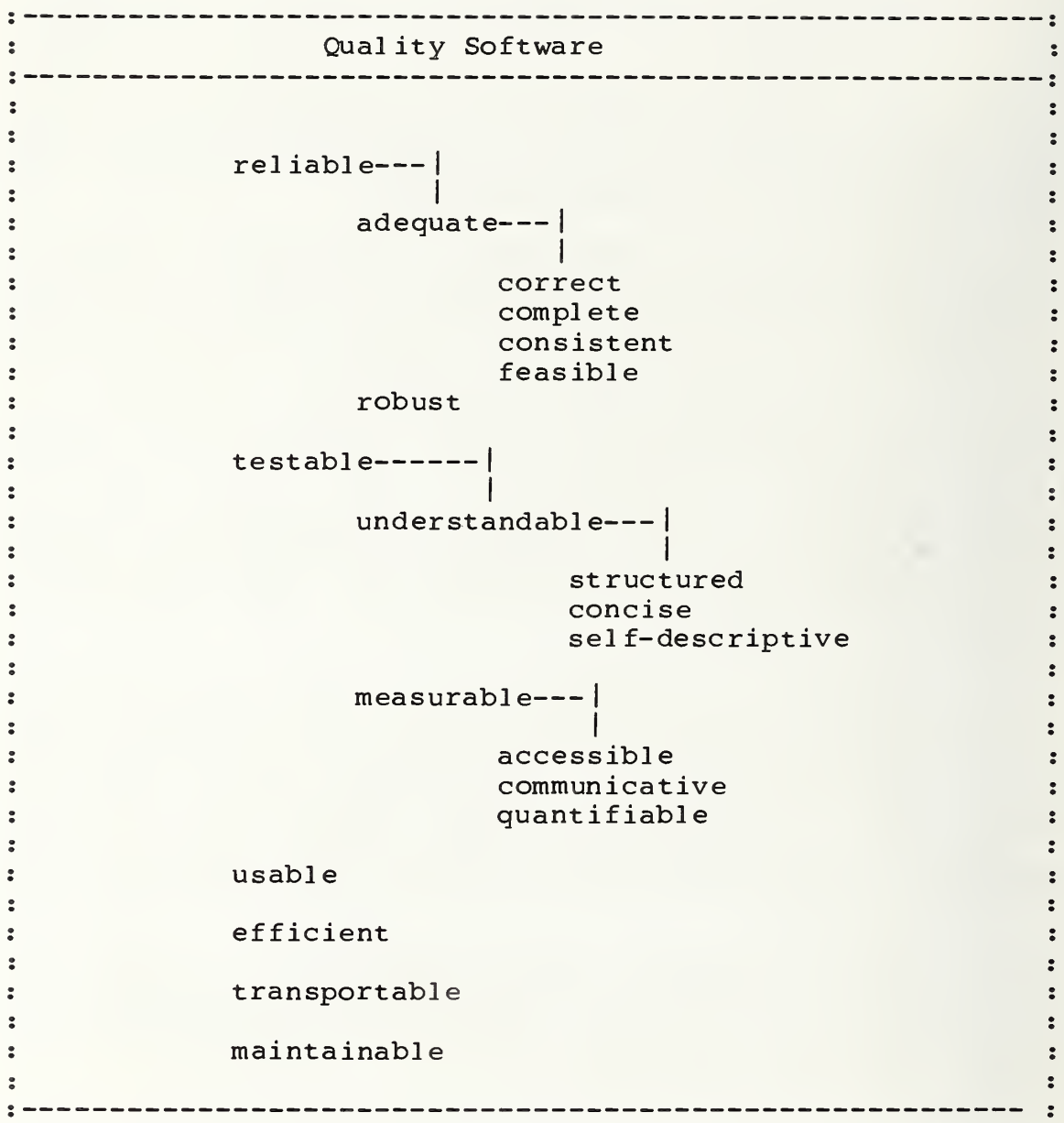


Figure 1.1 A Hierarchy of Software Quality Attributes

Even if good quality is difficult to define and measure, poor quality is glaringly apparent. Software that is error prone or does not work is obviously poor quality software. Consequently, discovery of errors in the software has been the first step toward quality assurance. Program testing, 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 inadequate. Consequently, increased emphasis has been placed upon insuring quality through the development process.

The criticality of the problem determines the effort required to validate the solution. Software to control airplane landings or to direct substantial money transfers requires higher confidence in its proper functioning than does a carpool locator program since the consequences of malfunction are more severe. For each software project not only the product requirements but also the validation requirements should be determined and specified at the initiation of the project. Project size, uniqueness, criticality, the cost of malfunction, and project budget all influence the validation needs. With the validation requirements clearly stated, specific techniques for verification and testing can be chosen. This document surveys the field of verification and testing techniques. The emphasis is upon medium and large size projects but many of the individual techniques have broader applicability. Verification and testing for very small projects are discussed in [BRAN80].

Although a glossary is included as an appendix to this document, the following terms are sufficiently important to warrant definition in the text. It should be noted that some of these terms may appear with slightly different meanings elsewhere in the literature.

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 lifecycle.

2. 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.

3. VERIFICATION: in general the demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the development lifecycle.

4. TESTING: examination of the behavior of a program by executing the program on sample data sets.

5. PROOF OF CORRECTNESS: use of techniques of logic to infer that an assertion assumed true at program entry implies that an assertion holds at program exit.

6. PROGRAM 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.

1.2 Verification Throughout the Lifecycle

Figure 1.2 presents a traditional view of the development life cycle with testing contained in 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. It is not unusual to hear of testing consuming 50% of the development budget. All errors are costly but the later in the lifecycle that the error discovery is made, the more costly the error [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 on software projects today is to defer the activity of detecting and correcting software problems until late in the project. The primary reason for early investment in verification activity is that expensive errors may already have been made before coding begins.

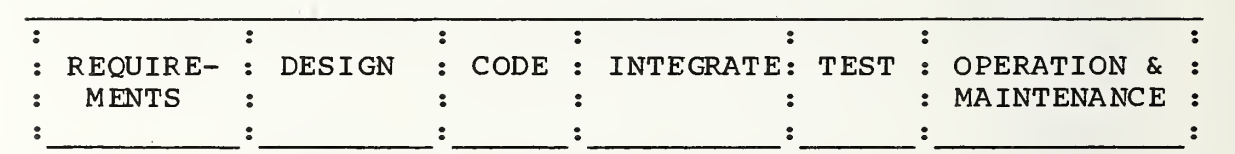


Figure 1.2 The Software Development Life Cycle

Figure 1.3 presents an amended life cycle chart which includes verification activities. The success of phasing verification throughout the development cycle depends upon the existence of a clearly defined and stated product at each development 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 firm product from the early development stages.

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 & Maintenance	Retest

Figure 1.3 Life Cycle Verification Activities

We will examine each stage of the lifecycle and discuss the relevant activities. The following activities should be performed at each stage:

1. Analyze the structures produced at this stage for internal testability and adequacy.

2. Generate test sets based on the structures at this stage.

In addition, the following should be performed during design and construction:

3. Determine that the structures are consistent with structures produced during previous stages.

4. Refine or redefine test sets generated earlier.

Throughout the entire life cycle, neither development nor verification is a straightline activity. Modifications or corrections to structures at one stage will require modifications and reverification of structures produced during previous stages.

REQUIREMENTS.

The verification activities that accompany the problem definition and requirements analysis stage of software development are extremely significant. The adequacy of the requirements must be thoroughly analyzed and initial test cases generated with the expected (correct) responses. Developing scenarios of expected system use may help to determine the test data and anticipated results. These tests will form the core of the final test set. Generating these tests and the expected behavior of the system clarifies the requirements and helps guarantee that they are testable. Vague or untestable requirements will leave the validity of the delivered product in doubt. Late discovery of requirements inadequacy 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 automated tools to aid in the requirements definition exist. Examples include Information System Design and Optimization System (ISDOS) [TEIC77], The Software Requirements Engineering Program (SREP) [ALFO77], Structured Analysis and Design Technique (SADT) [ROSS77], and Systematic Activity Modeling Method (SAMM) [LAMB78]. All provide a disciplined frame work for expressing requirements and thus aid in the checking of consistency and completeness. Although these tools provide only rudimentary validation procedures, this capability is greatly needed and it is the subject of current research [TEIC78].

DESIGN.

Organization of the verification effort and test management activities should be closely integrated with preliminary design. The general testing strategy, including test methods and test evaluation criteria, is formulated; and a test plan is produced. If the project size or criticality warrants, an independent test team is organized. In addition, a test schedule with observable milestones is constructed. At this same time, the framework for quality assurance and test documentation should be established, such as [FIPS76], [BUCK79], or [IEEE79].

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 test organization and 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 should be used by the developers to verify the flow and logical structure of the system while design inspection should be performed by the test team. Missing cases, faulty logic, module interface mismatches, data structure inconsistencies, erroneous I/O assumptions, and user interface inadequacies are items of concern. The detailed design must be shown internally consistent, complete, and consistent with the preliminary design and requirements.

Although much of the verification must be performed manually, the use of a formal design language can facilitate the analysis. Several different design methodologies 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. These techniques are manual and facilitate verification by providing a clear statement of the design. The Design Expression and Configuration Aid (DECA) [CARP75], the Process Design Language [CAIN75], High Order Software [HAMI76], and SPECIAL [ROUB76] are examples of automated systems or languages which can also be used for analysis and consistency checking.

CONSTRUCTION.

Actual testing 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 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 are used to provide further quality assurance. These techniques are discussed in detail in Chapter 3.

During the entire test process, careful control and management of test information is critical. Test sets, test results, and test reports should be catalogued and stored in a data base. 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.

MAINTENANCE.

Over 50% of the life cycle costs of a software system are spent on maintenance. As the system is used, it is modified either to correct errors or to augment the original system. After each modification the system must be retested. Such retesting activity is termed regression testing. The goal of regression testing is to minimize the cost of system revalidation. Usually only those portions of the system impacted by the modifications are retested. However, changes at any level may necessitate retesting, reverifying and updating documentation at all levels below it. For example, a design change requires design reverification, unit retesting and subsystem and system retesting. Test cases generated during system development are reused or used after appropriate modifications. The quality of the test documentation generated during system development and modified during maintenance will affect the cost of regression testing. If test data cases have been catalogued and preserved, duplication of effort will be minimized.

We will emphasize testing, verification, and validation during software development. The maintenance and operation stage is very important, but generally outside the scope of this report. The procedures described here for software development will, if followed correctly, make the task of maintaining, upgrading, evolving, and operating the software a much easier task.

2.1 Concepts

The purpose of this section is to discuss the basic concepts and fundamental implications and limitations of testing as a part of software verification. There are many meanings attributed to the verb "to test" throughout the technical literature. Let us begin by looking at the Oxford English Dictionary definition:

> Test - That by which the existence, quality, or genuineness of anything is, or may be, determined.

The objects that we test are the elements that arise during the development of software. These include modules of code, requirements and design specifications, data structures, and any other objects that are necessary for the correct development and implementation of our software. We will often use the term "program" in this document to refer to any object that may be conceptually or actually executed. A design or requirements specification can be conceptually executed, transforming input data to output data. Hence, remarks directed towards "programs" have broader application.

We view a program as a representation of a function. The function describes the relationship of an input (called a domain element) to an output (called a range element). The testing process is then used to ensure that the program faithfully realizes the function. For example, consider the function 1/x. Its domain is the set of all floating point numbers excluding Ø. Any program that realizes the function 1/x must, when given a floating point value r (r nonzero), return the value 1/r (given the machine dependent precision). The testing problem is to ensure that the program does represent the function.

Elements of the function's domain are called valid inputs. Since programs are expected to operate reasonably on elements outside of a function's domain (called "robustness"), we must test the program on such elements. Thus any program that represents 1/x should be tested on the value Ø and perhaps also on meaningless data (such as strings) to ensure that the program does not fail catastrophically. These elements outside of the function's domain are called invalid inputs. How to choose these and other test input values is discussed in detail in 3.3.

The essential components of a program test are a description of the functional domain, the program in executable form, a description of the expected behavior, a way of observing program behavior, 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 to test 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 the test instance succeeds, otherwise we say the test instance fails.

Of the five necessary components in the testing process, the most difficult to obtain is a description of the expected behavior. Often ad hoc methods must be used to determine expected behavior. These methods include hand calculation, simulation, and other less efficient solutions to the same problem. What is needed is an oracle, a source which for any given input description can provide a complete description of the corresponding output behavior. We will discuss this process throughout Chapter 3.

2.2 Dynamic Testing

We can classify program test methods into dynamic and static analysis techniques. Dynamic analysis requires that the program be executed and, hence, involves the traditional notion of program testing, i.e. 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 does not usually involve actual program execution. Common static analysis techniques include such compiler tasks as syntax and type checking. We will first consider some general aspects of 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 if not infinite very large, so as to make the number of required test instances infeasible.

The solution is to reduce this potentially infinite exhaustive testing process to a finite testing process. This is accomplished by finding criteria for choosing representative elements from the functional domain. These criteria may reflect either the functional description or the program structure.

The subset of elements used 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 that "covers" the domain and is small enough to perform the testing process for each element in the set. The paper of Goodenough and Gerhart [GOOD75] presents 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 consistent provided that test sets T1 and T2 chosen by C are such that all test instances of T1 are successful exactly when all test instances of T2 are successful. A criterion C is said to be complete provided that it produces test sets that uncover all errors. These definitions lead to the fundamental theorem of testing which states:

> If there exists a consistent, 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 to be impossible to find consistent, complete test criteria except for the simplest cases [HOWD76]. The above just confirms that testing, especially complete testing, is a very difficult process. Examples of criteria that are used in practice for the selection of test sets include:

1. The elements of the test set reflect special domain properties such as extremal or ordering properties.

2. The elements of the test set exercise the program structure such as test instances insuring all branches or all statements are executed.

3. The elements of the test set reflect special properties of the functional description such as domain values leading to extremal function values.

2.3 Structural vs Functional Testing

The properties that the test set is to reflect are classified according to whether they are derived from a description of the program's function or from the program's internal structure. Test data generation based on functional analysis and on structural analysis is described in 3.3 and 3.4. Classifying the test data inclusion criteria given above, the first and the third are based on functional analysis criteria while the second is based on structural analysis criteria. Both structural and functional analysis should be performed to insure adequate testing. Structural analysis-based test sets tend to uncover errors that occur during "coding" of the program, while functional analysis-based test sets tend to uncover errors that occur in implementing requirements or design specifications.

Although the criterion for generating a structure-based test set is normally simple, the discovery of domain elements that satisfy the criterion is often quite difficult. Test data are usually derived by iteratively refining the data based on the information provided by the application of structural coverage metrics. Since functional analysis techniques often suffer from combinatorial problems, the generation of adequate functional test data is no easier. As a result, ad hoc methods are often employed to locate data which stress the program.

2.4 Static Testing

The application of test data and the analysis of the results are dynamic testing techniques. The class of static analysis techniques is divided into two types: techniques that analyze consistency and techniques that measure some program property. The consistency techniques are used to insure program properties such as correct syntax, correct parameter matching between procedures, correct typing, and correct requirements and specification translation. The measurement techniques measure properties such as error proneness, understandability, and well-structuredness.

The simplest of the consistency checking static analysis techniques is the syntax checking feature of compilers. In modern compilers, this feature is frequently augmented by type checking, parameter matching (for modules), cross reference tables, static array bounds checking, and aliasing. Two advanced static analysis techniques are symbolic execution and program proving. The latter proves the consistency of stated relations between program variables before and after program segments. Symbolic execution performs a "virtual" execution of all possible program paths. Since an actual execution does not occur, the method is considered a static analysis technique. Both are described in detail in Chapter 3.

2.5 Manual vs Automated Testing

A final classification of methods can be made upon the basis of whether the method is a manual method such as structured walkthrough or code inspection, or whether the method is automated.

In Table 2.1 we list the verification methods that will be discussed throughout the rest of the paper. We provide a classification according to whether the method is dynamic or static, structural or functional, manual or automated. We also provide a reference to where the method is discussed in the body of the report.

TABLE 2.1 A SUMMARY OF TESTING TECHNIQUES

Technique	Section	Manual/ Automatic	Static/ Dynamic	Structural/ Functional
Correctness proof	3.1	both	static	both
Walkthroughs	3.1	manual	dynamic	both
Inspections	3.1	manual	static	both
Design reviews and audits	3.1	manual	static	both
Simulation	3.1	automated	dynamic	functional
Desk checking	3.1	manual	both	structural
Peer review	3.1	manual	both	structural
Executable specs.	3.2	automated	dynamic	functional
Exhaustive Testing	3.3	automated	dynamic	functional
Stress testing	3.3	manual	dynamic	functional
Error guessing	3.3	manual	dynamic	functional
Cause effect graphing	3.3	both	dynamic	functional
Design based functional testing	3.3	manual	dynamic	functional
Coverage based metric testing	3.4	automated	both	structural
Complexity based metric testing	3.4	automated	both	structural
Compiler based analysis	3.6	automated	static	structural
Data flow analysis	3.6	automated	static	structural
Control flow analysis	3.6	automated	static	structural
Symbolic execution	3.6	automated	static	structural
Instrumentation	3.7	automated	dynamic	structural
Combined techniques	3.8	automated	both	both

A description of verification, validation, and testing techniques can be arranged in several different ways. In keeping with the emphasis on verification throughout the lifecycle, we first present general techniques which span the stages of the lifecycle. The remaining sections are organized along the lines of the usual testing plan, providing discussion of test data generation, test data evaluation, testing procedures and analysis, and the development of support tools. Each procedure will be briefly discussed, with emphasis given to its role in the validation process and its advantages and limitations.

3.1 General Verification Techniques

Techniques that can be used throughout the lifecycle are described here. The majority of the techniques involve static analysis and can be performed manually. They can be utilized without a large capital expenditure, although for analysis of large systems automated aids are advised. These include traditional informal methods of desk checking and review, disciplined techniques of structured walkthroughs and inspections, and formal methods of proof of correctness. In addition, the role simulation plays in validation and verification is described.

DESK CHECKING AND PEER REVIEW.

Desk checking is the most traditional means for analyzing a program. It is the foundation for the more disciplined techniques of walkthroughs, inspections and reviews. In order to improve the effectiveness of desk checking, it is important that the programmer thoroughly review the problem definition and requirements, the design specification, the algorithms and the code listings. In most instances, desk checking is used more as a debugging technique than a testing technique. Since seeing one's own errors is difficult, it is better if another person does the desk checking. For example, two programmers can trade listings and read each others code. This approach still lacks the group dynamics present in formal walkthroughs, inspections, and reviews.

Another method, not directly involving testing, which tends to increase overall quality of software production is peer review. There are a variety of implementations of peer review [MYER79], but all are based on a review of each programmer`s code. A panel can be set up which reviews sample code on a regular basis for efficiency, style, adherence to standards, etc. and which provides feedback to the individual programmer. Another

possibility is to maintain a notebook of required "fixes" and revisions to the software and indicate 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 peer review panel.

WALKTHROUGHS, INSPECTIONS, AND REVIEWS.

Walkthroughs and inspections are formal manual techniques which are a natural evolution of desk checking. While both techniques share a common philosophy and similar organization, they are quite distinct in execution. Furthermore, while they both evolved from the simple desk check discipline of the single programmer, they use very disciplined procedures aimed at removing the major responsibility for verification from the developer.

Both procedures require a team, usually directed by a moderator. The team includes the developer, but the remaining 3-6 members and the moderator should not be directly involved 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 walkthrough lies in the conduct of the meeting. Both methods require preparation and study by the team members, and scheduling and coordination by the team moderator.

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. Guidance for developing the test criteria can be found in [MYER79], [FAGA76] and [WEIN71]. The developer is usually required to narrate the reading of the product. Many errors are found by the developer just by the simple act of reading aloud. Others, of course, are determined as a result of the discussion with team members and by applying the test criteria.

Walkthroughs differ from inspections in that the programmer does not narrate a reading of the product by the team, but 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 walkthrough is to encourage discussion, not just to complete the system simulation on the test data. Most errors are discovered through questioning the developer's decisions at various stages, rather than through the application of the test data.

At the problem definition stage, walkthrough and inspection can be used to determine if the requirements satisfy the

testability and adequacy measures as applicable to this stage in the development. If formal requirements are developed, formal methods, such as correctness techniques, may be applied to insure adherence with the quality factors.

Walkthroughs and inspections should again be performed at the preliminary and detailed design stages. Design walkthroughs and inspections will be performed for each module and module interface. Adequacy and testability of the module interfaces are very important. Any changes which result from these analyses will cause at least a partial repetition of the verification at both stages and between the stages. A reexamination of the problem definition and requirements may also be required.

Finally, the walkthrough and inspection procedures should be performed on the code produced during the construction stage. Each module should be analyzed separately and as integrated parts 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:

1. System 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. 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.

PROOF OF CORRECTNESS TECHNIQUES.

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. The program is "proved" if whenever the input assertion is true for particular values of variables and the program executes, then it can be shown that the output assertion is true for the possibly changed values of the program's variables. The issue of termination is normally treated separately.

There are two approaches to proof of correctness: formal proof and informal proof. A formal proof consists of developing a mathematical logic consisting of axioms and inference rules and defining a proof to be either a proof tree in the natural deduction style [GENT35] or to be a finite sequence of axioms and inference rules in the Hilbert-Ackermann style [CHUR56]. The statement to be proved is at the root of the proof tree or is the last object in the proof sequence. Since the formal proof logic must also "talk about" the domain of the program and the operators that occur in the program, a second mathematical logic must be employed. This second mathematical logic is usually not decidable.

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 lifecycle where a formal representation or description exists. The GYPSY [AMBL78] and HDM [ROBI79] methodologies use proof techniques throughout the development stages. HDM, for example, has as a goal the formal proof of each level of development. Good summaries of program proving and

correctness research are in [MANN74] and [ELSP72].

Heuristics for proving programs formally are essential but are not yet well enough developed to allow the formal verification of a large class of programs. In lieu of applying heuristics to the program, some approaches to verification require that the programmer provide information, interactively, to the verification system in order that the proof be completed. Examples include Gerhart's AFFIRM [GERH80] and Constable's PL/CV [CONS78]. Such information may include facts about the program's domain and operators or facts about the program's intended function.

A typical example of a program and its assertions is given below. The input assertion states that the program's inputs are respectively a non-negative integer and a positive integer. The output assertion states that the result of the computation is the smallest non-negative remainder of the division of the first input by the second. This example is due to Dijkstra and appears in [DIJK72].

Input Assertion {a >= 0 and d > 0 and integer(a) and integer(b)}

```
Program     integer r,dd;
            r := a; dd:=d;
            while dd<=r do dd:=2*dd;
            while dd~=d do
               begin dd:=dd/2;
                  if dd<=r do r:=r-dd
               end
```

Output Assertion {0 <= r < d and a congruent to r modulo d}

Informal proof techniques follow the logical reasoning behind the formal proof techniques but without the formal logical system. Often the less formal techniques are more palatable to the programmers. The complexity of informal proof ranges from simple checks such as array bounds not being exceeded, to complex logic chains showing non-interference of processes accessing common data. Informal proof techniques are always used implicitly by programmers. To make them explicit is similar to imposing disciplines, such as structured walkthrough, on the programmer.

SIMULATION.

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 nonreal-time applications in which simulation is a cost effective verification and test data generation technique.

To use simulation as a verification tool several models must be developed. Verification is performed by determining if the model of the software behaves as expected on models of the computational and external environments using simulation. This technique also is a powerful way of deriving test data. Inputs are applied to the simulated model and the results recorded for later application to the actual code. This provides an "oracle" for testing. The models are often "seeded" with errors to derive test data which distinguish these errors. The data sets derived cause errors to be isolated and located as well as detected during the testing phase of the construction and integration stages.

To develop a model of the software for a particular stage in the development lifecycle a formal representation compatible with the simulation system is developed. This may consist of the formal requirements specification, the design specification, or the actual code, as appropriate to 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.

The next steps are to develop a model of the computational environment in which the system will operate, a model of the hardware on which the system will be implemented, and a model of the external demands on the total system. These models can be largely derived from the requirements, with statistical representations developed for the external demand and the environmental interactions. The software behavior is then simulated with these models to determine if it is satisfactory.

Simulating the system at the early development stages is the only means of determining 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 interpretive 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 tradeoffs, the simulation under real loads does provide error information.

3.2 Test Data Generation

Test data generation is the critical step in testing. Test data sets must contain not only input to exercise the software, but must also provide the corresponding correct output responses to the test data inputs. Thus the development of test data sets involves two aspects: the selection of data input and the determination of expected response. Often the second aspect is most difficult. As discussed previously, hand calculation and simulation are two techniques used to derive expected output response. For very large or complicated systems, manual techniques are unsatisfactory and insufficient.

One promising direction is the development of executable specification languages and specification language analyzers [SRS79], [TEIC77]. These can be used, as simulation is used, to act as an oracle 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, when tested, is error free. This is a method often used in high density 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 occuring in hardware. This gap is closing with the advances in VLSI.

There is another statistical testing procedure for hardware that applies to certain software applications. Often integrated circuits are tested against a standard "correct" chip using statistically derived test sets. Applications of this technique include testing mass produced firmware developed for microcomputers embedded in high volume production devices such as ovens, automobiles, etc. A second possibility is to use this concept to test "evolving" software. For the development of an upwardly compatible operating system, some of the test sets can be derived by using a current field tested system as an oracle. Compiler testing employs a similar test set for each different compiler tested. However, since most software is developed as a "one of a kind" item, this approach generally does not apply.

... 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. No consideration is usually given to the internal organization, logic, control, or data flow in developing test data sets based on functional analysis. One technique, design-based functional analysis, 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 logical organization of the software. Most structural analysis is supported by test coverage metrics such as path coverage, branch coverage, etc. These criteria provide a measure of completeness of the testing process.

3.3 Functional Testing Techniques

The most obvious and generally intractable functional testing procedure is exhaustive testing. As was described in Chapter 2, only a fraction of programs can be exhaustively tested since the domain of a program is usually infinite or infeasibly large and cannot be used as a test data set. To attack this problem, characteristics of the input domain are examined for ways of deriving a representative test data set which provides confidence that the system will be fully tested.

As was stated in Chapter 2, test data must be derived from an analysis of the functional requirements and include representive 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, non-extremal, 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.

BOUNDARY VALUE ANALYSIS.

The problem of deriving test data sets is to partition the program domain in some meaningful way so that input data sets which span the partition can be determined. There is no direct, easily stated procedure for forming this partition. It 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 lifecycle.

At the requirements stage a coarse partitioning is obtained according to the overall functional requirements. At the design

stage, additional functions are introduced which define the separate modules allowing for 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 following references [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. Test data should be chosen which lie both inside each input class and at the boundary of each class. Output classes should also be covered by input which causes output at each class boundary and within each class. These data are the extremal and non-extremal 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. The array should be tested as a single element array A(1,1) and as a full 100 x 100 array. The array element values A(I,J) should be chosen to exercise the corresponding boundary values for each element. Examples for more complex data structures can be found in [HOWD79], [MYER79].

ERROR GUESSING AND SPECIAL VALUE ANALYSIS.

Myers suggests that some people have a natural intuition for test data generation [MYER79]. While this ability cannot be completely described nor formalized, certain test data seem highly probable to catch errors. Some of these are in the category Howden [HOWD80] calls special, others are certainly boundary values. Zero input values and input values which cause zero outputs 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 which are numerically very close and values which are numerically quite different. Guessing carries no guarantee for success, but neither does it carry any penalty.

CAUSE EFFECT GRAPHING.

Cause effect graphing [MYER79] is a technique for developing test cases for programs from the high level specifications. A high level specification of requirements states desired characteristics of behavior for the system. These characteristics can be used to derive test data. Problems arise, however, of a combinatorial nature. For example, a program that has specified responses to eight characteristic stimuli (called causes) given some input has potentially 256 "types" of input (ie. 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 methodical 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 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, for they cause the same effect. This analysis results in a partitioning of the causes according to their corresponding effects.

A limited entry decision table is then constructed from the directed graph reflecting these dependencies (ie. causes 2 and 3 result in effect 4, causes 2, 3, and 5 result in effect 6, etc.). The decision table is then reduced [METZ77] and test cases chosen 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.

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 [HOWD8Ø]. A distinction can be made between requirements functions and design functions. Requirements functions describe the overall functional capabilities of a program. In order to implement a requirements function it is usually necessary to invent other "smaller functions". These other functions are used 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. All functional capabilities represented by boxes at the second level in the tree correspond to design functions. The implementation of a design function may require the invention of other design functions. This successive refinement during top down design can then be represented as levels in the tree structure, where the n+1st level nodes are refinements or subfunctions of the nth level functions.

To utilize design based functional testing, the functional design trees as described above are constructed. The trees

document the functions used in the design of the program. The functions included in the design trees must be chosen carefully. The most important selection feature is that the function be accessible for independent testing. It must be possible to apply the appropriate input values to test the function, to derive the expected values for the function, and to observe the actual output computed by the code implementing the function.

Each of the functions in the functional design tree, if top down design techniques are followed, 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 even a single statement. 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 proceeds as described in the boundary value analysis discussion above. Extremal, non-extremal, and special values test data should be selected for each input variable. Test data should also be selected which results in the generation of extremal, non-extremal, and special output values.

3.4 Structural Testing Techniques

Structural testing is concerned with ensuring sufficient testing of the implementation of a function. Although used primarily during the coding phase, structural analysis should be used in all phases of the lifecycle where the software is represented formally in some algorithmic, design or requirements language. The intent of structural testing is to stress the implementation by finding 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: complexity-based testing or coverage-based testing. In the latter, a criterion is used which provides a measure of the number of structural units of the software which are fully exercised by the test data sets. In the former category, tests are derived in proportion to the software complexity.

COVERAGE-BASED TESTING.

Most coverage metrics are based on the number of statements, branches, or paths in the program which 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. The control flow is represented by directed lines or edges which connect the nodes. 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.

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 edges 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 caused. A program with variable loop control may have effectively an infinite number of paths. Hence, there are potentially an infinite number of path expressions.

To completely test the program structure, the test data chosen should cause the execution of all paths. Since this is not possible in general, metrics have been developed which give a measure of the quality of test data based on the proximity to this ideal coverage. Path coverage determination is further complicated by the existence of infeasible paths. Often a program has been inadvertently designed so that no data will cause the execution of certain paths. 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 which are simple and of low complexity so that all paths, excluding loop iteration, may be tested and that infeasible paths may be avoided.

All techniques for determining coverage metrics are based on graph representations 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 will 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 supply information about which statements have been executed (in addition to the percentage of coverage), the results can guide the selection of test data to insure complete coverage. To apply the metric, the program or module is instrumented by hand or by a preprocessor. A postprocessor or manual analysis of the results reveal the level of statement coverage. Determination of an efficient and complete test data set satisfying this metric is more difficult. Branch predicates that send control to omitted statements should be examined to help determine input data that will cause execution of omitted statements.

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. Segment coverage guarantees statement coverage. It also covers branches with no executable statements, eg. an IF-THEN-ELSE with no ELSE statements still requires data causing the predicate to be evaluated as both true and false. Techniques similar to those used for statement coverage are used for applying the metric and deriving test data.

The next logical step is to strengthen the metric by requiring separate coverage for both the exterior and interior of loops. Segment coverage only requires that both branches from a branch predicate be taken. For loops, segment coverage can be satisfied by causing the loop to be executed one or more times (interior test) and then causing the loop to be exited (exterior test). Requiring that all combinations of predicate evaluations be covered requires that each loop be exited without interior execution for at least one data set. This metric requires more paths to be covered than segment coverage requires. Two successive predicates will require at least four sets of test data to provide full coverage. Segment coverage can be satisfied by two tests, while statement coverage may require only one test for two successive predicates.

Implementation of the above metric is again similar to that for statement and segment coverage. Variations on this 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. This 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] [LYON74] [RAMA74] [MAIT80]. These tools are generally applicable to most of the coverage metrics described above. Automation of test data generation is less advanced. Often test data are generated by iterating the use of analyzers with 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] and [HOWD77]. More on the use of these tools will be discussed in a later section.

COMPLEXITY-BASED TESTING.

Several complexity measures have been proposed recently. Among these are cyclomatic complexity [MCCA76], software science [HALS77], and Chapin's software complexity measure [CHAP79]. These and many other metrics are designed to analyze the

complexity of software systems. Most, although valuable new approaches to the analysis of software, are not suitable, 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 are calculated from the program graph, while the third is a runtime metric.

McCabe uses a property of graph theory in defining cyclomatic complexity. 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 be easily calculated from the program graph.

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, C, must 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 a proper subgraph with only one node which is single entry and whose entering edge is not in the subgraph, and with only one node which is single exit and whose exiting edge is 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 based on counting these single entry, single exit proper subgraphs of two nodes or greater. Let the number of these subgraphs be m, then essential complexity EV(G) is defined:

$$EV(G) = CV(G) - m$$

The program graph for a program built with structured constructs will obviously have all proper subgraphs as single exit, single

entry. The number of proper subgraphs of a graph G of more than one node is CV(G)-1. Hence the essential complexity of a structured program is one. Essential complexity is then a measure of the "unstructuredness" of a program.

Actual complexity, AV, is just the number of paths executed during a run. A testing strategy can be based on these metrics. If for a test data set, the actual complexity is less than the cyclomatic complexity and all edges have been executed, then either there are more paths to be tested or the complexity can be reduced by CV(G)-AV by eliminating decision nodes and reducing portions of the program to in-line code. The cyclomatic complexity metric gives the number of linearly independent paths from analysis of the program graph. Some of these paths may be infeasible. If this is the case, then the actual complexity will never reach the cyclomatic complexity. Using a tool [MAIT80] which derives the three complexity metrics, both a testing and a programming style can be enforced.

3.5 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% to 95%. It is much more difficult to find test data providing 90% coverage under the more complex coverage metrics. However, it has been noted [BROW73] that methodologies based on the more complex metrics with lower coverage requirements have uncovered as many as 90% of all program faults.

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 from an analysis of the 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. There are two things we would like to know, a maximum liklihood estimate for the number of errors and a level of confidence measure on that estimate. The technique is to insert known errors in the code in some way that is statistically similar to the actual errors. The test data is then applied and the number of known seeded errors and the number of original errors uncovered is determined. In [MILL72] it is noted that, if one assumes that the statistical properties of the seeded and original errors is the same and that the testing and seeding are statistically unbiased, then

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.

How good is this estimate? We would like to ascertain the confidence level for the various predicted error levels. Again from [MILL72], assuming that all seeded errors are detected (K=S), the confidence that number of errors is less than or equal to E is given by:

$$\begin{cases} 0 & ; I > E \\ \dfrac{S}{S + E + 1} & ; I <= E \end{cases}$$

More elaborate formulae for the case that all seeded errors are not found, and for cases where partial results are known are given in [MILL72] and [TAUS77].

Note that when E = 0 and no errors are detected other than seeded errors (I<=E) when testing, the confidence level is very high (for S= 99, confidence = 99%). Testing for the error free case can be accomplished with high confidence as long as no errors are uncovered. On the other hand, if nonseeded errors are discovered and the estimate for E is higher, our confidence in the estimate also decreases. If the E = 10, then with S = 100, our confidence drops to 90%. When the number of actual errors approaches or exceeds the number of seeded errors, then the confidence in our estimates decreases dramatically. For example, if E = 10 and S = 9, then the confidence is only 45%.

A strategy for using this statistical technique in dynamic testing is to monitor the maximum liklihood estimator, and perform the confidence level calculation as testing progresses. If the estimator gets high relative to the number of seeded errors, then it is unlikely that a desirable confidence level can be obtained. The errors should then be corrected and the testing resumed. If the number of real errors discovered remains small or preferably zero as the number of seeded errors uncovered approaches the total seeded, then our confidence level for an error free program increases.

Tausworthe [TAUS77] discusses a method for seeding errors which has some hope of being similar statistically to 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 actual modifications of the code are similar to those used in mutation testing as described below.

MUTATION ANALYSIS.

A relatively new metric developed by DeMillo, Lipton, and Sayward is called mutation analysis [DEMI78]. This method rests on the competent programmer hypothesis which states that a program written by a competent programmer will be, after debugging and testing, "almost correct." The basic idea of the method is to seed the program to be tested with errors, creating several mutants of the original program. The program and its mutants are then run interpretively on the test set. If the test set is adequate, it is argued, it should be able to distinguish between the program and its mutants.

The method of seeding is crucial to the success of the technique and consists of modifying single statements of the program in a finite number of "reasonable" ways. The developers conjecture a coupling effect which implies that these "first order mutants" cover the deeper, more subtle errors which might be represented by higher order mutants. The method has been subject to a small number of trials and so far has been successfully used interactively to develop adequate test data sets. It should be noted that the method derives both branch coverage and statement coverage metrics as special cases.

It must be stressed that mutation analysis, and its appropriateness, rests on the competent programmer and coupling effect theses. Since neither is provable, they must be empirically demonstrated to hold over a wide variety of programs before the method of mutations can itself be validated.

3.6 Static Analysis Techniques

As was described in Chapter 2, analytical techniques can be categorized as static or dynamic. The application and analysis of test data is usually described as a dynamic activity, since it involves the actual execution of code. Static analysis does not usually involve program execution. Many of the general techniques discussed in 3.1, such as formal proof techniques and inspections are static analysis techniques. In a true sense, static analysis is part of any testing technique. Any analysis to derive test data, calculate assertions, or determine instrumentation breakpoints must involve some form of static analysis, although the actual verification is achieved through dynamic testing. As was mentioned in Chapter 3, the line between static and dynamic analysis is not always easily drawn. For example, proof of correctness and symbolic execution both "execute" code, but 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 such as code optimization, listing of variable names, and pretty printing, all such activities being useful in the production of quality software. 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 software shop coding standards can be enforced, quality of code can be monitored, and adherence to programming standards (such as FORTRAN77 [ANSI78]) can be checked.

FLOW ANALYSIS.

Data and control flow analysis are similar in many ways. Both are based upon graphical representation. In control flow analysis, the program graph has nodes which represent a statement or segment possibly ending 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 in other 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. The technique was introduced by Cocke and Allen [ALLE74], [ALLE76] for global program optimization.

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

```
SUBROUTINE HYP(A,B,C)
U = Ø.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. It is possible 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. The problem is not in detecting these errors, but in resolving them. The possible solution suggested may not be the correct one. There is no answer to this problem, 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 while the program executes. This behavior can be classified by when a particular variable is referenced, defined, or undefined in the program. A variable is referenced when its value must be 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, such as when a variable appears on the left hand side of an assignment. A variable is 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 by associating, at each node in the data flow graph, values for tokens (representing program variables) which indicate whether the corresponding variable is referenced, unreferenced, or defined with the execution of the statement represented by that node. If symbols, for instance 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. A typical path expression might be drlllllrrlllllldllrllu, which can be reduced through eliminating nulls to drrrdru. Such a path expression contains no anomalies, but the presence of ...dd... in an expression, indicating a variable defined twice without being referenced, does identify a potential anomaly. Most anomalies, ..ur.., r..., etc. 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,

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

The variable X is certainly referenced in the statement, but it may be defined only if the predicate is true. In such a case, two nodes would be used, and the graph would actually represent code which looked like

```
        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 after which the index variable becomes undefined only if the loop is exited. Subroutine and function calls introduce further problems, but they too can be resolved. The use of data flow analysis for static analysis and testing is described in [OSTE76] and [FOSD76].

SYMBOLIC EXECUTION.

Symbolic execution is a method of symbolically defining data that force program paths to be executed. Instead of executing the program with actual data values, the variable names that hold the input values are used. Thus all variable manipulations and decisions are made symbolically. As a consequence, all variables become string variables, all assignments become string assignments and all decision points are indeterminate. To illustrate, consider the following small pseudocode program:

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

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

```
if a*a + b = Ø then x := Ø else if a*a + b ~= Ø 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. The expression 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 effectively taken.

If the program has no loops, then the resultant tree structure is finite. The tree structure can then 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 conjunction. Data that causes a particular path to be executed can be found by determining which data will make the path conjunction 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.

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. 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 conjunct has values which satisfy it is undecidable even with restricted programming languages. For certain applications, however, the method has been successful.

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 the first statement of the loop and at the last statement of the loop (thus the assertion remains "invariant" over one execution of the loop). From this assertion, an assertion true before entrance to the loop and assertions true after exit of the loop can be constructed. The program can then be viewed as "free" of loops (ie. each loop is considered as a single statement) and assertions extended to all statements of the program (so it is fully annotated) using techniques similar to the backward substitution method described above for symbolic execution. A good survey of these methods appears in [HANT76] and examples of their use in verifiers appear in [LUCK79] and [GERH80].

3.7 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 turnstyle or a counter. Branch or segment coverage and other such metrics are evaluated in this manner. A preprocessor analyzes the program (usually by generating a program graph) and inserts counters at the appropriate places. Consider

```
            IF (X) 10,10,15
            .
            .
10          Statement i
            .
            .
```

```
15      Statement j
        .
        .
        DO 20 I = J,K,L
        .
        .
20      Statement k
```

A preprocessor might instrument the program segment as follows:

```
        IF (X) 100,101,15
        .
        .
100     N(100) = N(100) + 1
        GO TO 10
101     N(101) = N(101) + 1
10      Statement i
        .
        .
15      N(15) = N(15) + 1
        Statement j
        .
        .
        I = J
        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
```

For the IF statement, each possible branch was instrumented. Note that we used two counters N(100) and N(101) even though the original code branches to the same statement label. The original code had to be modified for the DO loop in order to get the necessary counters inserted. Note that two counters are used, N(20) for the interior execution count and N(201) for the exterior of the loop.

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

Instrumentation does not have to rely on direct code

insertion. Often calls to runtime routines are inserted rather than actual counters. Some instrumented code is passed through a preprocessor/compiler which inserts the instrumentation only if certain 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 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 using simple commands and 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(extended-logical-expression)[optional
                                          qualifier][control]
```

The optional qualifiers are ALL, SOME, etc. The control options include LEVEL, which controls the levels in a block structured program; CONDITIONS, which allows dynamic enabling of the instrumentation; and LIMIT, which allows a specific number of violations to occur. The logical expression is used to represent an expected condition to be dynamically verified. For example:

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

placed 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 that 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. Note the similarity with verification conditions, especially if the entry-exit assertions are employed. Furthermore, symbolic execution can be employed to generate the assertions as it can be used with proof techniques. Some efforts are currently underway to integrate dynamic assertions, proof techniques, and symbolic evaluation. One of these is described below.

There are many other techniques for dynamic analysis. Most involve the dynamic (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.

3.8 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 is the merger of standard testing techniques with formal verification. Our ability, through formal methods, to verify significant segments of code is improving [GERH78], and moreover there are certain modules, which for security or reliability reasons, justify the additional expense of formal verification.

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

Another example, combining dataflow analysis, symbolic execution, elementary theorem proving, dynamic assertions, and standard testing is suggested in [OSTE8Ø]. 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. Here, paths are executed symbolically so that each decision point and every loop has an assertion. The assertions are then checked for consistency using both dataflow and proof techniques. If all the assertions along a path are consistent, then they can be reduced to a single dynamic assertion for the path. Theorem proving techniques can be employed to "prove" the path assertion and termination, or the path can be tested and the dynamic assertions evaluated for the test data.

The technique allows for several tradeoffs between testing and formal methods. For instance, symbolically derived dynamic assertions are more reliable than manually derived assertions, but cost more to generate. Consistency analysis of the assertions using proof and dataflow techniques adds cost at the front end, but reduces the execution overhead. Finally there is the obvious tradeoff between theorem proving and testing to verify the dynamic assertions.

3.9 Test Support Tools

Testing, like program development, generates large amounts of information, necessitates numerous computer executions, and requires coordination and communication between workers. Support tools and techniques can ease the burden of test production, test execution, general information handling, and communication. General system utilities and text processing tools are invaluable

for test preparation, organization, and modification. A well organized and structurable file system and a good text editor are a minimum support set. A more powerful support set includes data reduction and report generation tools. Library support systems consisting of a data base management system and a configuration control system are as useful during testing as during software development since data organization, access, and control are required for management of test files and reports. Documentation can be viewed as a support technique. In addition to the general purpose support tools and techniques, specific test support tools exist. Test drivers and test languages are in this category. The following paragraphs will discuss these test specific support tools and techniques.

TEST DOCUMENTATION.

FIPS PUB 38 [FIPS76], the NBS guideline for software documentation during the development phase, recommends test documentation be prepared for all multipurpose or multiuser projects and for other software development projects costing over $5000. FIPS 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 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 functions of agency ADP management practice and will vary depending upon the size, complexity, and risk of the project.

TEST DRIVERS.

Unless the module being developed is a stand-alone program, considerable auxiliary software must be written in order to exercise and test it. Auxiliary code which sets up an appropriate environment and calls the module is termed a driver while code which 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.

When testing is performed incrementally, an untested module is combined with a tested one and the package is then tested. Such packaging can lessen the number of drivers and/or stubs which must be written. When the lowest level of modules, those which 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. This approach is called bottom-up testing. Bottom-up testing still requires that test drivers be constructed. Testing which starts with the executive module and incrementally adds modules which it calls, is termed top-down testing. Top-down testing requires that stubs

be created to simulate the actions of called modules that have not yet been incorporated into the system. The testing order utilized should be coordinated with the development methodology used.

AUTOMATIC TEST SYSTEMS AND TEST LANGUAGES.

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, an effort has been made to automate the process of test execution. Programs that perform this function of initiation are called test drivers, test harnesses, or test systems.

The simplest test drivers merely reinitiate the program with various input sets and save the output. The more sophisticated test systems accept data inputs, expected outputs, the names of routines to be executed, values to be returned by called routines, and other parameters. These test systems not only initiate the test runs but compare the actual output with the expected output and issue concise reports of the performance. TPL/2.Ø [PANZ78] which uses a test language to describe test procedures is an example of such a system. In addition to executing the tests, verifying the results and producing reports, the system helps the user generate the expected results.

PRUFSTAND [SNEE78] is an example of a comprehensive test system. It is an interactive system in which data values are generated automatically or are requested from the user as they are needed. The system is comprised of a:

. preprocessor to instrument the code

. translator to convert the source data descriptors into an internal symbolic test data description table.

. test driver to initialize and update the test environment

. test stubs to simulate the execution of called modules

. execution monitor to trace control flow through the test object

. result validator

. test file manager

. post processor to manage reports

A side benefit of a comprehensive test system is that it establishes a standard format for test materials, which is

extremely important for regression testing. Currently automatic test driver systems are expensive to build and consequently are not in widespread use.

In the previous sections 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 walkthroughs, reviews, and inspections, applied to all stages in the lifecycle ([FACA76]). 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 key verification techniques.

For the construction stage, automated techniques can be of great value. The ones in widest use are the simpler static analysis techniques (such as type checking), automated test coverage calculation, automated program instrumentation, and the use of simple test harnesses. These techniques are relatively straightforward to implement and all have had broad use. Combined with careful error documentation, they are effective validation methods.

Many of the techniques discussed in chapter 3 have not seen wide use. The principal reasons for this include their specialization (simulation), the high cost of their use (symbolic execution), and their unproven applicability (formal proof of correctness). Many of these techniques represent the state of the art in program validation and are in areas where research is continuing.

The areas showing the most commercial interest and activity at present include automated test support systems and increased use of automated analysis. As more formal techniques are used during requirements and design, an increase in automatic analysis is possible. In addition, more sophisticated analysis techniques are being applied to the code during construction. More complete control and automation of the actual execution of tests, both in assistance in generating the test cases and in the management of the testing process and results, are also taking place.

We re-emphasize the importance of performing validation throughout the lifecycle. 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. A disciplined approach must be followed during both planning and execution of the verification activities.

We view the integration of validation with software development as so important that 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 acceptability criteria which are to be used during software development. These requirements are in addition to the functional requirements of the system ordinarily specified at this stage. Thus embedded within the project requirements would be a contract aimed at enhancing the quality of the completed software.

A major difficulty with a proposal such as the above 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; the goal is to try to get as close as required to perfect. In addition, what constitutes perfect and how important it is for the software to be perfect may vary from project to project. Some software (such as reactor control systems) needs to approach perfection more closely than other software (such as an address labeling program). The definition of "perfect" (or quality attributes) and its importance should be part of the validation requirements. However, validation mechanisms written into the requirements do not guarantee high quality 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 second difficulty with specifying a collection of validation methods in the requirements is that most validation tools do not exist in integrated packages. This means that the group performing the verification must learn several tools that may be difficult to use in combination. This is a problem that must receive careful thought. For unless the combination is chosen judiciously, their use can lead to additional costs and errors. The merits of the tool collection as a whole must be considered as well as the usefulness of any single tool.

Future work in validation should address the above issues. One possible course of action is to integrate the development and validation techniques into a "programming environment". Such an environment would encompass the entire software development effort and include verification capabilities to:

1. Analyze requirements and specifications

2. Analyze and test designs

3. Provide support during construction (e.g. test case generation, test harnesses)

4. Provide a data base sufficient to support regression testing

The use of such environments has the potential to improve greatly the quality of the completed software and also to provide a mechanism for establishing confidence in the quality of the software. 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.

AUDIT: see DOD DEVELOPMENT REVIEWS

BLACK BOX TESTING: see FUNCTIONAL TESTING.

BOUNDARY VALUE ANALYSIS: a selction 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 3.3)

BRANCH TESTING: a test method satisfying coverage criteria that require that for each decision point each possible branch be executed at least once. (see 3.4)

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 which will cover the entire effect set. (see 3.3)

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 at least one condition to be true. (see 2.2)

CONSISTENT CONDITION SET: A set of boolean conditions such that complete test sets for the conditions uncover the same errors. (see 2.2)

CYCLOMATIC COMPLEXITY: The cyclomatic complexity of a program is equivalent to the number of decision statements plus 1. (see 3.4)

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 3.4)

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 3.3)

DOD DEVELOPMENT REVIEWS: A series of reviews required by DOD directives. System requirements review, system design review, preliminary design review, critical design review, functional configuration audit, physical configuration audit, and formal qualification review comprise the set of required life cycle reviews. (see 3.1)

DRIVER: code which sets up an environment and calls a module for test. (see 3.9)

DYNAMIC ANALYSIS: analysis that is performed by executing the program code. (see 3.7)

DYNAMIC ASSERTION: a dynamic analysis technique which inserts assertions about the relationship between program variables into the program code. The truth of the assertions is determined as the program executes. (see 3.7)

ERROR GUESSING: test data selection technique. The selection criterion is to pick values that seem likely to cause errors. (see 3.3)

EXHAUSTIVE TESTING: executing the program with all possible combinations of values for program variables. (see 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 3.3)

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 3.3)

INFEASIBLE PATH: a sequence of program statements that can never be executed. (see 3.4)

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 3.1)

INSTRUMENTATION: The insertion of additional code into the program in order to collect information about program behavior during program execution. (see 3.7)

INVALID INPUT (TEST DATA FOR INVALID INPUT DOMAIN): test data that lie outside the domain of the function the program represents. (see 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 1.2)

METRIC BASED TEST DATA GENERATION: the process of generating test sets for structural testing based upon use of complexity metrics or coverage metrics. (see 3.4)

MUTATION ANALYSIS: A method to determine test set thoroughness by measuring the extent to which a test set can discriminate the program from slight varients (mutants) of the program. (see 3.5)

ORACLE: a mechanism to produce the "correct" responses to compare with the actual responses of the software under test. (see 2.1)

PATH EXPRESSIONS: a sequence of edges from the program graph which represents a path through the program. (see 3.4)

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 clases; one path from each class is then tested. (see 3.4)

PRELIMINARY DESIGN REVIEW: see DOD DEVELOPMENT REVIEWS.

PROGRAM GRAPH: graphical representation of a program. (see 3.4)

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 3.1)

REGRESSION TESTING: testing of a previously verified program required following program modification for extension or correction. (see 1.2)

SIMULATION: use of an executable model to represent the behavior of an object. During testing the computional hardware, the external environment, and even code segments may be simulated. (see 3.1)

SELF VALIDATING CODE: code which makes an explicit attempt to determine its own correctness and to proceed accordingly. (see 3.7)

SPECIAL TEST DATA: test data based on input values that are likely to require special handling by the program. (see 3.3)

STATEMENT TESTING: a test method satisfying the coverage criterion that each statement in a program be executed at least once during program testing. (see 3.4)

STATIC ANALYSIS: analysis of an program that is performed without executing the program. (see 3.6)

STRESS TESTING: see BOUNDARY VALUE ANALYSIS.

STRUCTURAL TESTING: a testing method where the test data are derived solely from the program structure. (see 3.4)

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 3.9)

SYMBOLIC EXECUTION: a static analysis technique that derives a symbolic expression for each program path. (see 3.6)

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 2.1)

TEST DRIVER: a program which 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 3.9)

TEST HARNESS: see TEST DRIVER.

TESTING: examination of the behavior of a program by executing the program on sample data sets. (see 2.1)

VALID INPUT (TEST DATA FOR A VALID INPUT DOMAIN): test data that lie within the domain of the function represented by

the program. (see 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 lifecycle.

VERIFICATION: in general the demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the development lifecycle.

WALKTHROUGH: a manual analysis technique in which the module author describes the module's structure and logic to an audience of colleagues. (see 3.1)

WHITE BOX TESTING: see STRUCTURAL TESTING.

REFERENCES

[ADRI80] W. R. Adrion and P. M. Melliar-Smith, "Designing for the Unexpected," Computer to appear (1980).

[ALFO77] M.W.Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Transactions on Software Engineering, vol SE-2, no 1, pp 60-69 (1977).

[ALLE74] F. E. Allen, "Interprocedural Data Flow Analysis," Proceedings IFIP congress 1974, North-Holland Publishing Co., Amsterdam, pp398-402 (1974).

[ALLE76] F. E. Allen and J. Cocke, "A Program Data Flow Procedure," Communications of the ACM, vol. 19, no.3, p137-147 (March 1976).

[AMBL78] A.L.Ambler, D.I.Good, J.C.Browne, W.F.Burger, R.M.Cohen, C.G.Hoch and R.E.Wells, "Gypsy: A Language for Specification and Implementation of Verifiable Programs," Proceedings of an ACM Conference on Language Design for Reliable Software, D.B.Wortman,ed., New York, pp 1-10 (1978).

[ANSI78] ANS FORTRAN X3.9-1978, American National Standards Institute, New York (1978).

[BAKE72] F.T.Baker, "Chief Programmer Team Management of Production Programming," IBM Systems Journal, vol 11, no 1, pp 56-73 (1972).

[BELL77] T.E.Bell, D.C.Bixler and M.F.Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Transactions on Software Engineering, vol SE-3, no 1, pp 49-60 (1977).

[BOEH78] B.W.Boehm, J.R.Brown, H.Kaspar, M.Lipow, G.J.MacLeod and M.J.Merrit, Characteristics of Software Quality, North- Holland, Amsterdam-New York-Oxford, 1978.

[BOEH77] B.W.Boehm, "Seven Basic Principles of Software Engineering," Software Engineering Techniques, Infotech State of the Art Report, (1977).

[BOYE75] R.S.Boyer, B.Elspas and K.N.Levitt, "SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution," Proceedings of 1975 International Conference on Reliable Software, pp. 234-245 (1975).

[BRAN8Ø] M. A. Branstad, J. C. Cherniavsky, and W. R. Adrion, "Validation, Verification, and Testing for the Individual Programmer," Computer ,Vol. 13, No. 12, (Dec. 198Ø).

[BROW73] J.R. Brown, et al., "Automated Software Quality Assurance," PROGRAM TEST METHODS W. Hetzel, Ed.,Prentice-Hall,Englewood Cliffs, Chapter 15 (1973).

[BUCK79] F. Buckley, "A Standard for Software Quality Assurance Plans," Computer vol. 12,no. 8, pp43-5Ø (August 1979).

[BUDD78a] T.Budd, R.A.DeMillo, R.J.Lipton and F.G.Sayward, "The Design of a Prototype Mutation System for Program Testing," AFIPS Conference Proceedings, vol 47, 1978 Computer Conference, pp 623-627 (1978).

[BUDD78b] T. A. Budd and R. J. Lipton, "Mutation Analysis of Decision Table Programs," Proceedings of the 1978 Conference on Information Science and Systems, Johns Hopkins University, pp346-349 (1978).

[CAIN75] S. H. Caine and E. K. Gordon, "PDL: A Tool for Software Design," Proceedings of the National Computer Conference, AFIPS Press, Montdale, N. J., (1975).

[CARP75] L.C.Carpenter and L.L.Tripp, "Software Design Validation Tool," 1975 International Conference on Reliable Software, April 1975.

[CHAP79] N.Chapin, "A Measure of Software Complexity," Proceedings of the AFIPS National Computer Conference, pp. 995-1ØØ2, June,1979.

[CHUR56] A.Church, Introduction to Mathematical Logic, Vol. 1, Princeton University Press, Princeton, 1956.

[CONS78] R. L. Constable and M. J. O'Donnell, A Programming Logic, Winthrop Publishing Co., Cambridge (1978).

[DEMI78] R.A.DeMillo, R.J.Lipton and F.G.Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," Computer, vol 11, no 4, pp 34-43 (1978).

[DIJK72] E.W.Dijkstra, "Notes on Structured Programming," in Structured Programming, O.J.Dahl, E.W.Dijkstra and C.A.R.Hoare, eds., Academic Press, London, 1972 (pp 1-82).

[ELSP72] B.Elspas, K.N.Levitt, R.J.Waldinger and A.Waksman,

"An Assessment of Techniques for Proving Program Correctness, Computing Surveys, vol 4, no 2, pp 97-147 (1972).

[FAGA76] M.E.Fagan, "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, vol 15, no 3, pp 182-211 (1976).

[FIPS76] "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., February, 1976.

[FOSD76] L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability," Computing Surveys vol. 8,no. 3, pp305-330 (September 1976).

[GENT35] G.Gentzen, "Investigations into Logical Deductions," in The Collected Works of Gerhard Gentzen, M.E. Szabo ed., North- Holland, Amsterdam, 1969 (pp 68-128).

[GERH78] S. L. Gerhart, "Program Verification in the 1980's: Problems, Perspectives, and Opportunities," ISI/RR-78-71, Information Sciences Institute, Marina del Rey (August 1978).

[GERH80] S. L. Gerhart, D. R. Musser, D. H. Thompson, D. A. Baker, R. L. Bates, R. W. Erickson, R. L. London, D. G. Taylor, and D. S. Wile, "An Overview of AFFIRM; A Specification and Verification System, Proceedings IFIP Congress 1980, North- Holland Publishing Co., Amsterdam, to appear (1980).

[GOOD75] J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, Vol. SE-1, no. 2, (1975).

[HALS77] M.H.Halstead, Elements of Software Science, Elsevier North-Holland, New York, 1977.

[HAMI76] N.Hamilton and S.Zeldin, "Higher Order Software - A Methodology for Defining Software," IEEE Transactions on Software Engineering, vol SE-2, no 1, pp 9-32 (1976).

[HANT76] S. L. Hantler and J. C. King, "An Introduction to Proving the Correctness of Programs," Computing Surveys, vol. 8, no. 3, p331-53 (1976).

[HOWD76] W.E.Howden, "Reliability of the Path Analysis Testing Strategy," IEEE Transactions on Software

Engineering, vol SE-2, no. 3, (1976).

[HOWD77] W.E.Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Transactions on Software Engineering, vol SE-3, no 4, pp 266-278 (1977).

[HOWD78] W.E.Howden, "A Survey of Dynamic Analysis Methods," in Tutorial: Software Testing and Validation Techniques, E.Miller and W.E.Howden, ed., IEEE Computer Society, New York, 1978.

[HOWD80] W. E. Howden, "Functional Program Testing," IEEE Transactions on Software Engineering, vol. SE-6, no. 2, pp162-169 (1980).

[INFO79] Software Testing, INFOTECH State of the Art Report Infotech, London (1979).

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

[JACK75] M.A.Jackson, Principles of Program Design, Academic Press, New York, 1975.

[JONE76] C. Jones, "Program Quality and Programmer Productivity," IBM Technical Report, International Business Machines Corp., San Jose, (1976).

[KERN74] B. W. Kernighan, "RATFOR- A Preprocessor for a Rational FORTRAN," Bell Labs Internal Memorandum, Bell Laboratories, Murray Hill (1974).

[KOPP76] R. Koppang, "Process Design System- An Integrated Set of Software Development Tools," Proceedings of the Second International Software Engineering Conference, October 1976.

[LAMB78] S.S.Lamb, V.G.Leck, L.J.Peters and G.L.Smith, "SAMM: A Modeling Tool for Requirements and Design Specification," Proceedings COMPSAC 78, IEEE Computer Society, New York, 1978 (pp 48-53).

[LIPT78] R.J.Lipton and F.G.Sayward, "The Status of Research on Program Mutation," Proceedings of The Workshop on Software Testing and Test Documentation, IEEE Computer Society, New York, pp 355-367, 1978.

[LUCK79] D. Luckham, S. German, F. von Henke, R. Karp, P.

Milne, D. Oppen, W. Polak, and W. Schenlis, "Stanford Pascal Verifier User's Manual," AI Memorandum CS-79-731, Stanford University, Stanford (1979).

[LYON74] G.Lyon and R. B. Stillman, "A FORTRAN Analyzer," NBS Technical Note No. 849, National Bureau of Standards, Washington (1974).

[MAIT80] R. Maitland, "NODAL," in "NBS Software Tools Database," R. Houghton and K. Oakley, editors, NBSIR-xx, National Bureau of Standards, Washington (1980).

[MANN74] Z.Manna, Mathematical Theory of Computation, McGraw- Hill, New York, 1974.

[MCCA76] T.J.McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, vol SE-2, no 4, pp. 308-320 (1976).

[MCCA77] J. McCall, P. Richards, and G. Walters, "Factors in Software Quality," volumes 1-3, NTIS AD-A049-014,015,055 (1977).

[METZ77] J. R. Metzner and B. H. Barnes Decision Table Languages and Systems Academic Press, New York (1977).

[MILL70] H.D.Mills, "Top Down Programming in Large Systems," in Debugging Techniques in Large Systems, R. Rustin ed., Prentice- Hall, 1970 (pp 41-55).

[MILL72] H. D. Mills, "On statistical validation of computer programs," IBM Report FSC72-6015, Federal Systems Division, IBM, Gaithersburg, Md., 1972.

[MILL75] E.F.Miller,Jr., "RXVP - An Automated Verification System for FORTRAN," Proceedings Workshop 4, Computer Science and Statistics: Eighth Annual Symposium on the Interface, Los Angeles, California, February 1975.

[MILL77] E.F.Miller,Jr., "Program Testing: Art Meets Theory," Computer, vol 10, no 7, pp 42-51 (1977).

[MILS76] "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] G.J.Myers, Software Reliability - Principles and Practices, John Wiley and Sons, New York (1976).

[MYER79] G. J. Myers, The Art of Software Testing, John

Wiley and Sons, New York (1979).

[NEUM75] P. G. Neumann, L. Robinson, K. Levitt, R. S. Boyer, A. R. Saxema, "A Provably Secure Operating System," SRI Project 2581, SRI International, Menlo Park (1975).

[OSTE76] L. J. Osterweil and L. D. Fosdick, "DAVE- A Validation, Error Detection, and Documentation System for FORTRAN programs," Software Practice and Experience, vol. 6, pp473-486 (1976).

[OSTE80] L. J. Osterweil, " A Strategy for Effective Integration of Verification and Testing Techniques," Technical Report CU- CS-181-80, University of Colorado, Boulder (1980).

[PANZ78] D.J. Panzl, "Automatic Revision of Formal Test Procedures," Third International Conference on Software Engineering, May 1978.

[PARN77] D.L. Parnas, "The Use of Precise Specifications in the Development of Software," Information Processing 77, B. Gilchrist, editor, North Holland, (1977).

[RAMA74] C.V.Ramamoorthy and S.F.Ho, Fortran Automated Code Evaluation System, ERL - M466, University of California, Berkeley, California, 1974.

[ROBI79] L. Robinson, "The HDM Handbook, Volume I-III," SRI Project 4828, SRI International, Menlo Park (1979).

[ROSS77] D.T.Ross and K.F.Schoman,Jr., "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, vol SE-3, no 1, pp 6-15 (1977).

[ROUB76] O.Roubine and L.Robinson, Special Reference Manual, Stanford Research Institute Technical Report CSG-45, Menlo Park, Calif.,1976.

[TAUS77] Robert C. Tausworthe, Standardized Development of Computer Software, Prentice-Hall, Englewood Cliffs, N. J., 1977.

[SNEE78] H. Sneed and K. Kirchoff, "Prufstand- A Testbed for Systematic Software Components," in Proceedings INFOTECH State of the Art Conference on Software Testing, Infotech, London (1978).

[SRS79] Proceedings of the Specifications of Reliable Software Conference IEEE Catalog No. CH1401-9C, IEEE, New

York (1979).

[STUC77] L. G. Stucki, "New Directions in Automated Tools for Improving Software Quality," in Current Trends in Programming Methodology, Volume II-Program Validation, R. Yeh, editor, Prentice-Hall, Englewood Cliffs, pp80-11 (1977)

[TAUS77] R. C. Tausworthe, Standardized Development of Computer Software, Jet Propulsion Laboratory, Pasadena, (1978).

[TEIC77] D.Teichroew and F.A.Hershey,III, "PSL/PSA: A Computer- Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, vol SE-3, no 1, pp 41-48 (1977).

[TEIC78] D. Teichroew personal communication.

[WEIN71] G.M.Weinberg, The Psychology of Computer Programming, Van Nostrand Reinhold, New York, 1971.

[WHIT78] 1. J. White and E. I. Cohen, " A Domain Strategy for Computer Program Testing," Digest for the Workshop on Software Testing and Test Documentation, Ft. Lauderdale, pp335-354 (1978).

[YOUR79] E.Yourdon and L.L.Constantine, Structured Design, Prentice-Hall, Englewood Cliffs, N.J., 1979.

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See instructions)	1. PUBLICATION OR REPORT NO. NBS SP 500-75	2. Performing Organ. Report No.	3. Publication Date February 1981

4. TITLE AND SUBTITLE

Validation, Verification, and Testing of Computer Software

5. AUTHOR(S)

W. Richards Adrion, Martha A. Branstad, John C. Cherniavsky

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

Same as above.

10. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 80-600199

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

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 programming product. Verification throughout the development process is stressed. Specific tools and techniques are described.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

Automated software tools; software lifecycle; software testing; software verification; test coverage; test data generation; validation; verification

13. AVAILABILITY

☒ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☒ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☐ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

62

15. Price

$3.75

NBS TECHNICAL PUBLICATIONS

PERIODICALS

JOURNAL OF RESEARCH—The Journal of Research of the National Bureau of Standards reports NBS research and development in those disciplines of the physical and engineering sciences in which the Bureau is active. These include physics, chemistry, engineering, mathematics, and computer sciences. Papers cover a broad range of subjects, with major emphasis on measurement methodology and the basic technology underlying standardization. Also included from time to time are survey articles on topics closely related to the Bureau's technical and scientific programs. As a special service to subscribers each issue contains complete citations to all recent Bureau publications in both NBS and non-NBS media. Issued six times a year. Annual subscription: domestic $13; foreign $16.25. Single copy, $3 domestic; $3.75 foreign.

NOTE: The Journal was formerly published in two sections: Section A "Physics and Chemistry" and Section B "Mathematical Sciences."

DIMENSIONS/NBS—This monthly magazine is published to inform scientists, engineers, business and industry leaders, teachers, students, and consumers of the latest advances in science and technology, with primary emphasis on work at NBS. The magazine highlights and reviews such issues as energy research, fire protection, building technology, metric conversion, pollution abatement, health and safety, and consumer product performance. In addition, it reports the results of Bureau programs in measurement standards and techniques, properties of matter and materials, engineering standards and services, instrumentation, and automatic data processing. Annual subscription: domestic $11; foreign $13.75.

NONPERIODICALS

Monographs—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NBS, NBS annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a worldwide program coordinated by NBS under the authority of the National Standard Data Act (Public Law 90-396).

NOTE: The principal publication outlet for the foregoing data is the Journal of Physical and Chemical Reference Data (JPCRD) published quarterly for NBS by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements available from ACS, 1155 Sixteenth St., NW, Washington, DC 20056.

Building Science Series—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The standards establish nationally recognized requirements for products, and provide all concerned interests with a basis for common understanding of the characteristics of the products. NBS administers this program as a supplement to the activities of the private sector standardizing organizations.

Consumer Information Series—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.

Order the **above** *NBS publications from: Superintendent of Documents, Government Printing Office, Washington, DC 20402.*

Order the **following** *NBS publications—FIPS and NBSIR's—from the National Technical Information Services, Springfield, VA 22161.*

Federal Information Processing Standards Publications (FIPS PUB)—Publications in this series collectively constitute the Federal Information Processing Standards Register. The Register serves as the official source of information in the Federal Government regarding standards issued by NBS pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations).

NBS Interagency Reports (NBSIR)—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Services, Springfield, VA 22161, in paper copy or microfiche form.

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Washington, D.C. 20234

OFFICIAL BUSINESS

Penalty for Private Use, $300

POSTAGE AND FEES PAID
U.S. DEPARTMENT OF COMMERCE
COM-215

U.S.MAIL

SPECIAL FOURTH-CLASS RATE
BOOK