UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-82-135	2. GOVT ACCESSION NO. AD A118254	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SOFTWARE TESTING MEASURES		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 1 Jun 80 - 30 Sep 81
		6. PERFORMING ORG. REPORT NUMBER CR-4-993
7. AUTHOR(s) W. Heidler A. Kerbel J. Benson A. Pyster R. Meeson		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0188
9. PERFORMING ORGANIZATION NAME AND ADDRESS General Research Corporation P O Box 6770 Santa Barbara CA 93111		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55811823
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COEE) Griffiss AFB NY 13441		12. REPORT DATE May 1982
		13. NUMBER OF PAGES 362
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Frank S. LaMonica (COEE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software testing
Computer program testing
Testing and vericiation techniques
Test tools

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report examines the current state of development of automated software testing techniques. The report identifies and describes techniques that are useful for detecting errors in software. It also examines techniques for proving the correctness of programs, for debugging (locating and correcting errors), and for producing documentation automatically. The techniques are evaluated in the areas of effectiveness, reliability, cost, and ease of use--criteria for each of these

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

categories was developed as a part of the study effort. Profiles are presented for five major categories of test techniques--each profile describes in detail the capabilities of a technique, the automated tools that support it, the types of errors that it can detect, its degree of dependence on user skill and judgment, its applicability to various types of software, and its costs in terms of analysis time and computer resources. Important features and shortcomings of the techniques are discussed. The appendices to the report include: a set of guidelines for testing software; a survey of available automated tools which support the techniques, an automated bibliography of testing, and a description and results of an experiment with assertion testing.

Accession For
NTIS GRA&I [✓]
DTIC TAB ☐
Unannounced ☐
Justification

By
Distribution/
Availability Codes

Dist	Avail and/or Special
A	

DTIC COPY INSPECTED 2

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

CONTENTS

TABLES

TABLES

1 INTRODUCTION AND SUMMARY

This report examines the current state of development of automated software testing techniques. The report identifies and describes techniques that are useful for detecting errors in software. It also examines techniques for proving the correctness of programs, for debugging (locating and correcting errors), and for producing documentation automatically. The techniques are evaluated in the areas of effectiveness, reliability, cost, and ease of use--criteria for each of these categories were developed as a part of the study effort. Profiles are presented for five major categories of test techniques--each profile describes in detail the capabilities of a technique, the automated tools that support it, the types of errors that it can detect, its degree of dependence on user skill and judgment, its applicability to various types of software, and its costs in terms of analysis time and computer resources. Important features and shortcomings of the techniques are discussed. The appendices to the report include: a set of guidelines for testing software; a survey of available automated tools which support the techniques; an annotated bibliography of testing; and a description and results of an experiment with assertion testing.

1.1 EVALUATIONS OF THE TEST TECHNIQUES

Five general types of automated test techniques are treated in this study. Each technique is described in detail in Chapter 2. The five techniques are:

- Static analysis - a general term given to all procedures which check the syntax and semantics of a program without running the program. It includes such checks as data flow analysis, type checking, and standards checking.

- Executable assertions - logical expressions in the language of the program being tested. Assertions can check the values that variables in the program may assume, the relationship among variables, the flow of control, and the history of computation. Violations of assertions are reported to the user of the program through an error message.

- Structural testing - programs have natural groupings of structural elements, such as statements, branches, and paths. The purpose of structural testing is to increase the percentage of all of the structural elements in a program that are executed in a series of tests. Tools are used to determine whether a structural element is actually executed.

- Functional testing - a method for systematically developing test cases for a program. Test cases are selected on the basis of the functions that a program is supposed to perform and by analyzing the inputs to and output of the program.

- Formal techniques - symbolic execution and formal verification. Symbolic execution attempts to produce a mathematical expression for a program's output in terms of its input variables. Formal verification seeks a mathematically rigorous proof that a program satisfies its specifications.

In addition to these five techniques, there are tools available which provide assistance in debugging (locating and correcting errors) and in determining how efficiently a program uses computer resources. This report is mainly concerned with program testing, validation, and verification; however, the debugging and performance measurement capabilities of software tools are treated briefly in Chapters 2 and 6.

Chapter 3 presents measures developed for evaluating these techniques. The measures cover the general areas of effectiveness, reliability, cost, and ease of use. The <u>effectiveness</u> criteria are:

- The kinds of errors detected
- The percentage of total program errors detected

A reliable testing technique is one that is effective in detecting errors regardless of the conditions under which it is applied. In particular, effectiveness should not depend heavily on the abilities of the people applying the techniques.

A technique should also be applicable to as wide a range of testing environments as possible--batch and interactive programs, numerical and non-numerical applications, real-time and non-time-critical systems, etc. Therefore, the three reliability criteria which we have selected are:

- Insensitivity to human factors
- Reliability for various error types
- Insensitivity to characteristics of the program being tested

The cost criteria are:

- Analysis time required
- Computer resources
- Cost of tools

The ease of use criteria are:

- User skills required
- Degree of user involvement
- Analysis required for error detection and location

The literature on testing was searched extensively for data on which to base our evaluations. Several experiments on the effectiveness and costs of automated testing had previously been performed at GRC, and these were analyzed. Persons who had recently completed projects which used structural testing and formal verification were contacted and interviewed about their experiences. An experiment using executable assertions to test programs with real and artificially introduced errors was performed (Appendix D).

The evaluations are presented in Chapters 4 and 5. Chapter 4 compares the techniques in each category, using rank-orderings where appropriate. Table 1.1 shows these rankings. Chapter 5 contains profiles of each of the five techniques.

TABLE 1.1

RANKINGS OF THE FIVE TEST TECHNIQUES

(1 = best)

Technique	Range of Error Types Detected	Insensitivity to Human Factors	Overall Cost	Ease of Use
Static analysis	4	1	1	1
Executable assertions	2	4	3	4
Structural testing	3	2	2	2
Functional testing	1	3	4	3
Formal verification	-	5	5	5

Our conclusions from the evaluations can be summarized as follows:

- There is not enough data or experience with automated testing to determine a most effective technique. The widest range of errors can be detected by functional testing; the most limited range by static analysis. However, the techniques complement each other in important ways if they are used together.

- Static analysis is the most reliable technique, since it is highly automated and therefore least susceptible to human error. Functional testing, executable assertions, and the formal techniques all depend heavily on the skill of the tester in order to produce effective results.

- The formal techniques can be applied to the most narrow range of program types. Symbolic execution is feasible only for small segments of code. So far, formal verification has been successfully applied only to small programs and in proving the security properties of operating systems.

- Static analysis is the least expensive technique. It requires little work from the user, and in an efficient implemention requires overhead comparable to a compiler.

- Executable assertions and structural testing require similar amounts of computer resources. However, executable assertions also increase the amount of effort required to code a program.

- The costs of all of the dynamic test techniques (executable assertions and structural and functional testing) are very sensitive to the number of runs made of the program being tested. It is difficult to estimate the number of runs required to thoroughly test a program.

- The formal techniques are very expensive because they require a large amount of highly skilled work.

- Static analysis is the easiest technique to use. It is the most highly automated and requires the least skill on the part of the user.

- Executable assertions are rather difficult to use because it takes special skill and knowledge to write effective assertions.

- Structural testing does not require as much skill as functional testing or writing assertions, but provides little automated assistance in generating test data or detecting errors.

- Functional testing is not now highly automated, although there are many tools available to assist in various tasks. Test data generation is the biggest difficulty, since it must be done manually and there are no criteria of test thoroughness to guide the process.

Tables 1.2 through 1.5 summarize the available experience with using the four static and dynamic test techniques. Although the formal techniques have not been applied by software developers other than researchers in the field of formal verification, some successes have recently been achieved with them. These are described in Sec. 5.5.2.

TABLE 1.2

PROFILE SUMMARY - STATIC ANALYSIS

EFFECTIVENESS

Types of Errors Detected

Data handling errors
Interface errors
Data definition errors
Data base errors
Documentation errors

Percentage of Total Program Errors Detectable
(based on available data): 16% to 55%

RELIABILITY

Static analysis is highly automated, so error checking is applied consistently. Because it is automated, human error is unlikely to corrupt the testing process. But static testing has inherent limitations; there are few types of errors that it can consistently catch.

Static testing is usually most effective during the early stages of program development. Static analysis can be used with equal ease on large and small programs. The features and standards of new programming languages make some static checks unnecessary.

COST

Analysis Time - requires only the execution of a test tool and examination of its output. Saves time since it replaces manual functions.

Computer Resources - tool execution time is only two to four times compile time for an efficient tool. Storage costs vary according to the tool and computer system used.

TABLE 1.3
PROFILE SUMMARY - EXECUTABLE ASSERTIONS

EFFECTIVENESS

Types of Errors Detected
Computation errors
Logic errors
Data input errors
Data handling errors
Interface errors
Data definition errors
Data base errors

Percentage of Total Program Errors Detectable
(based on available data); up to 80%.

RELIABILITY

How reliable assertions are depends upon how well the person writing them understands the way his program is supposed to operate. Assertions have to be debugged just like the rest of a program.

The test data used has a great effect on the reliability of executable assertion testing. Test data must cause assertions to be violated or errors will go undetected.

Not all error conditions can be described in assertions, and sometimes only weak conditions can be imposed through assertions.

Assertions are useful throughout the software life cycle and on programs of all sizes.

COST

Analysis Time - Increases the coding and debugging effort according to the number of assertions used. Assertions may comprise anywhere from 5% to 50% of the total code.

Computer Resources - Also varies with the number of assertions used. Available data suggest the following estimates:

Increase in compilation time 5-125%
Increase in execution time: 0-40%
Increase in size of object code: 6-15%

TABLE 1.4

PROFILE SUMMARY - STRUCTURAL TESTING

EFFECTIVENESS

Types of Errors Detected
Computation errors
Logic errors
Data handling errors
Data output errors

Percentage of Total Program Errors Detectable
(based on available data): 20% - 90%

RELIABILITY

The test data used has a great deal of influence on the reliability of structural testing. Input data that tests boundary conditions or *singularities and demonstrates the operation of* program functions should be used when doing structural testing.

Structural testing is guaranteed to find errors only when a program path handles all input data incorrectly. Since this is not the case for all errors, structural testing alone cannot ensure that a program is operating correctly.

COST

Analysis Time - requires the user to generate test data and analyze output for errors. Experience suggests one-half to two days per error found.

Computer Resources - instrumentation tools generally require a 20% - 100% increase in object program size, and a 2% - 50% increase in execution time.

TABLE 1.5

PROFILE SUMMARY - FUNCTIONAL TESTING

EFFECTIVENESS

Types of Errors Detected
Computation errors
Logic errors
Data input errors
Data handling errors
Data output errors
Interface errors
Data definition errors
Data base errors
Operation errors
Documentation errors

Percentage of Total Program Errors Detectable
(based on available data): 50% - 90%

RELIABILITY

Functional testing requires the user to exercise skill and judgment in selecting test data and in determining the correctness of program output. Methodology advances have shown how to do effective functional testing, but not how to choose test data efficiently.

There is no data on the reliability of functional testing for different error types.

Functional testing works well under top-down program development, since functional capabilities are available early in the life cycle. Functional testing of large, complex programs can be difficult and error-prone. The technique has been proven effective for mathematical software, but problems of testing non-numeric programs have not been addressed.

COST

Analysis Time - this is the most significant cost involved in functional testing. Time is needed to generate test data and to examine the output for errors. Total costs depend on the number of test runs made.

Computer Resources - tools that support functional testing require very little overhead and can provide a significant cost savings over manual methods.

1.2 FURTHER ANALYSIS OF THE TEST TECHNIQUES

Besides detecting errors in programs, automated testing techniques can help the user to locate and correct errors. The techniques can also be beneficial in providing documentation and in evaluating the overall quality of a piece of software. These topics are discussed in Chapter 6.

Static analysis can provide a great deal of information about the location of errors. It can isolate errors in module interfaces, violations of coding standards, and instances of mixed-mode arithmetic to a single statement. Static analysis labels the type of error detected, which is useful information for correcting errors.

Assertion testing will locate errors to the code segment between the last branch point (i.e., IF or other decision statement) and the assertion violated, if a thorough set of assertions is used. Structural testing can only locate an error to an entire path through a program. Functional testing itself provides no error location information.

The test techniques should be used to retest a program after an error has been corrected. A metric of the efficiency of a technique for retesting is:

$$\frac{\text{(number of retesting runs)} \times \text{(amount of code exercised in one retesting run)}}{\text{amount of code changed}}$$

For static analysis this ratio is one test of one module per module changed. Functional and assertion testing used by themselves require all tests to be rerun when an error is found. Structural testing gives an indication of what code is affected by a test run, so that not all tests must be rerun.

Testing techniques can be used to provide program documentation and support the evaluation of software quality. Static analysis techniques supply much of the information for these and they could be enhanced to provide more.

In Sec. 6.4 we give a summary of the reports produced by static analysis and relate them to the program documentation reports required by MIL-STD-483. Assertions can also be useful in providing in-line program documentation, although they do not replace other documentation which is traditionally required.

Testing techniques can also be used to support the determination of overall software quality. Software quality has been defined by a set of desirable characteristics (maintainability, testability, etc.) in two studies, one by TRW[1] and the other by General Electric.[2] These characteristics are related to measurable properties of the software (number of comments, length of modules, etc.). The testing techniques can support the measurement of these properties in three ways:

- By directly measuring the property as part of the testing process.

- By providing information from which a measurement can be derived.

- By adding the property to the software during the process of testing.

[1] B. W. Boehm, et. al., Characteristics of Software Quality, TRW Systems Group Report No. TRW-SS-73-09, December 28, 1973.

[2] J. A. McCall, P. K. Richards, and G. Walters, Factors in Software Quality, Metric Data Collection and Validation, General Electric Co., under contract to Rome Air Development Center, Report No. RADC-TR-77-369, Vol. II, November 1977.

We found that the most properties were supported by static analysis. In all, we estimated that 42% of the TRW properties were supported by testing techniques and 32% of the GE properties were supported. Details of our analysis are presented in Sec. 6.5.

1.3 PROSPECTS FOR IMPROVEMENTS IN THE TEST TECHNIQUES

We expect each test technique to be improved in the future. The improvements will come as a result of advanced software engineering techniques—specifically the use of new programming languages—and from combining several techniques into integrated, comprehensive testing tools. This subject is discussed in Chapter 7.

Table 1.6 summarizes our expectations for improvements in the techniques. For each technique, the table lists current problems and anticipated solutions.

TABLE 1.6

TEST TECHNIQUE: PROBLEMS AND SOLUTIONS

Technique	Problems	Solutions
1. Static Analysis	Extraneous type error warnings	Improved programming languages
	Extraneous data flow error warnings	Libraries of global data flow information; algorithms to detect unexecutable paths
2. Executable Assertions	No guidelines for placing assertions	Heuristics for assertion placement and contents
	Difficulty in writing assertions	Language extensions for assertions
3. Structural Testing	No mechanism for error detection and reporting	Combine with executable assertions
	Requires test cases to be constructed manually	Use symbolic execution to generate test cases
4. Functional Testing	Requires testing separate program sections manually	Automated tools for independently testing program parts, keeping track of test data, and input regions tested
5. Formal Techniques	Poorly supported by automated tools	More powerful tools and better user interfaces
	Not widely accepted	More training in formal methods for programmers

2 AUTOMATED SOFTWARE TESTING TECHNIQUES

Automated software testing techniques have been classically separated into static analysis techniques and dynamic testing techniques. Tests that require analyzing only the program source code fall under static analysis techniques. Tests that require running the program fall under dynamic testing techniques. Newer techniques such as symbolic execution and program proving, however, blur this distinction.

Static, dynamic, and formal techniques are complementary testing methods. Each individual kind of analysis or specific run-time check covers a different (but usually overlapping) set of possible program errors. The kinds of common programming errors detected by these techniques are listed in the following discussion whenever a clear assessment can be made. The technique profiles in Chapter 5 provide more complete information on their effectiveness for a wide variety of errors.

2.1 SOURCE CODE STATIC AND STRUCTURAL ANALYSIS

Static analysis is a general term for all analysis and checking which requires only the program source code as input. For example, all the syntax checking done by a compiler would be included under static analysis. Additional static checks frequently not performed by compilers have been provided by automated test tools. These include tests for uninitialized variables, type conflicts, module interface conflicts, and incorrect parameter usage, to name just a few.[1]

Structural analysis is a subcategory of static analysis that deals with the flowchart-like structure of program source code. This kind of analysis can identify unreachable code, infinite loops,[2] and recursive

[1] Leon J. Osterweil, and Lloyd D. Fosdick, "DAVE — A Validation Error Detection and Documentation System for FORTRAN Programs," Software: Practice and Experience, Vol. 6, No. 4 (Oct-Dec. 1976).

[2] Here, the terms unreachable code and infinite loop refer to structural characteristics, and not the corresponding logical characteristics, which are more difficult to detect.

procedure calls.[1] It can also provide helpful documentation on the procedure-calling structure of large programs.

For this study of software testing measures the various static analysis techniques are grouped into five subcategories:

- Program error detection
- Program anomaly detection
- Assertion checking
- Test data generation
- Program documentation

Error detection techniques include techniques that will identify program conditions that cannot be interpreted correctly, i.e., "hard" errors. Anomaly detection techniques include techniques which provide warnings for program conditions that violate "good" usage and are likely to produce errors. Static assertions are a means to provide additional error checking. Test data generation and program documentation are two more ways static analysis supports software testing. Each of these categories is developed in detail below.

2.1.1 Detectable Program Errors

A number of program errors can be detected by automated (static) analysis of program source code. These errors are program conditions which invariably lead to incorrect computations. The principal examples of such errors are:

- Infinite loops - programs containing non-terminating computations.
- Module interface conflicts - mismatching actual and formal parameter specifications, including type conflicts and aliasing.

[1] Richard N. Taylor and Leon J. Osterweil, "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, pp. 265-278 (May 1980).

- Recursive procedure calls - procedures which directly or indirectly call themselves.[1]
- Uninitialized variables - attempt to use data which has not been defined.
- Deadlock - concurrent processes waiting for each other to respond.

<u>Infinite Loops</u>

Although some programs are supposed to run indefinitely (e.g., operating systems and embedded control programs), most programs and all individual modules should be free of non-terminating computations. Structured programming techniques have largely reduced this problem by restricting the use of the GOTO statement. In "unstructured" programs infinite loops can be detected by analyzing a flowchart-like graph of the program's organization. Any point in the program which cannot be reached by working backwards from the program's end is part of an infinite loop. This analysis is one of the structural analysis techniques.

In structured programs, DO-WHILE and REPEAT-UNTIL types of loops can be checked to see that variables in the loop exit conditions are modified within the body of the loop. This check does not guarantee loop termination but can identify certain errors that can lead to infinite loops such as incorrect exit conditions and missing statements in the loop body. Formal verification of loop termination is considered beyond the scope of most static analysis tools. Several high-level systems programming languages include LOOP or CYCLE statements for explicit coding of non-terminating loops where they are required.

[1]FORTRAN and COBOL do not support recursion.

Module Interface Conflicts

There are three primary static checks which can be made between procedure (and function) definitions and calling statements.

- Number of parameters
- Type correspondence between parameters
- Aliasing of global data

Conflicts in the number and type of parameters between procedure definitions and their calling statements are detected by maintaining a central database for module interface information. The actual arguments supplied in a calling statement can then be matched with the formal parameters from the procedure definition. Several common programming errors which can be detected by this kind of analysis include:

- Missing arguments in procedure (and function) call statements
- Extraneous arguments in calls
- Arguments listed out of order
- Wrong variable passed as an argument
- Wrong procedure called

The first two of these errors are detected by simply checking for the correct number of arguments. The second two can be detected when the datatypes of the actual and formal parameters do not match. The last error can often be detected using both of these techniques when the wrong procedure is called with parameters meant for the right one. Additional information can be stored in the database so that modifications to a procedure definition can be traced back to all calling statements to aid program maintenance.

Aliasing in computer programs is a condition where a single storage location can be referred to by more than one variable name. This is considered a programming error because of the side-effect of all aliased variables being modified by an assignment to any one of them. Additional aberrations appear when various different parameter passing methods are considered. The two primary sources of aliasing which can be detected by static analysis are:

- Global data passed as procedure arguments
- A single variable passed in two argument positions

Additional complexity results when array elements are potentially aliased. In this case warning messages are usually sufficient to get the programmer to verify the absence of aliasing conditions.

Recursive Procedure Calls

Complex processing tasks can often be described most concisely as recursive procedures or functions. However, several high-level programming languages, including FORTRAN and COBOL, do not support recursive procedure definitions. Recursive procedure calls can be identified by loops appearing in the calling graph structure of a program. In FORTRAN and COBOL programs, the calling graph structure must be loop-free. JOVIAL programs can be checked to see that only procedures declared with the "REC" attribute are used recursively.

Uninitialized Variables

References to program variables before meaningful values are assigned to them are considered to be in error. Many computer systems initialize all program variables automatically. However, programs which rely on this system-dependent feature will not run properly on other computer systems which do not perform (the same) initialization. Uninitialized variables are detectable by data flow analysis which determines the sequence of set and use operations on all program variables. Any set/use sequence which does not start with a set operation indicates an uninitialized variable.

Deadlock

Deadlock is a condition where concurrent processes become "stalled" waiting for one another to signal the continuation of a computation. When no progress can be made because all processes have reached a "wait" state, the system is said to be deadlocked. Two approaches to handling the problem of deadlocks are available: prevention, or avoidance. Deadlocks are prevented by ensuring that the conditions necessary for their occurrence cannot exist.[1] Unfortunately, deadlock prevention can often only be achieved at a significant cost in resource utilization. Methods of guaranteeing analytically that a system is deadlock-free have been developed for a few applications. Taylor and Osterweil[2] describe how static data flow analysis could be used to prevent a few special forms of deadlock.

Detection or avoidance of deadlock means incorporating fault-tolerance into a system. The way processes request and use resources is monitored dynamically. Algorithms are applied which determine that a deadlock has occurred or is about to occur. When a problem is found, corrective action must be taken. The merits of a few approaches to deadlock detection are discussed by Gligor and Shattuck.[3]

2.1.2 Detectable Program Anomalies

A somewhat larger class of abnormal or error-prone program constructs, which may or may not really be errors, can be identified by additional static analysis techniques. Because these program anomalies

[1] E. G. Coffman, M. J. Elphick and A. Shoshani, "System Deadlocks," Computing Surveys, Vol. 3, No. 2 (June 1971), pp. 67-78.

[2] R. N. Taylor and L. J. Osterweil, "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," IEEE Transactions on Software Engineering, Vol. SE-6, No. 3 (May 1980), pp. 265-278.

[3] V. D. Gligor and S. H. Shattuck, "On Deadlock Detection in Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-6, No. 5 (Sept. 1980), pp. 435-440.

do not necessarily affect correct computations, they are usually flagged with warning messages by automated tools. Program structures which are considered to be anomalous include:

- Violations of coding standards
- Mixed-mode expressions
- Data flow anomalies
- Unreachable coded
- Unreferenced statement labels

Each of these categories is elaborated in more detail below.

Coding Standards

Modern software engineering techniques for program development often restrict the use of certain constructs provided in high-level programming languages. The principal examples of this are the restrictions often placed on the use of the GOTO statement. Structured control statements provided in modern languages (or by pre-processors for older languages) have reduced the need for GOTOs to a few special circumstances, such as recovering from processing errors. For most practical purposes the GOTO can be relegated to the class of program anomalies and reported by static analysis warnings.

Additional examples of modern thinking on programming techniques which impose restrictions on the "free use" of facilities provided by programming languages include:

- Requiring declaration of all program variables so that no default characteristics are mistakenly inherited
- Limiting the size of program modules as a method of encouraging program modularity

- Limiting the depth of structured statement nesting to reduce module complexity
- Limiting the length of individual statements to reduce expression complexity

These restrictions allow the detection of errors caused by misspelling identifiers (since they will have been declared using a different spelling) and tend to reduce mistakes due to source code complexity. Psychological studies have shown, for example, that people have trouble interpreting expressions with more than five levels of parenthesis nesting.[1]

Mixed Mode Expressions

Computational expressions requiring the implicit conversion of program data from one type to another are called mixed-mode expressions. Modern software engineering philosophy says that such implicit conversions, like default attributes, are error-prone and should be avoided. In cases where datatype conversions are required, the explicit conversion operations are usually very simple. Where code can be rewritten to eliminate datatype conversions, it is often simplified. Errors which can be caught in some cases by disallowing mixed-mode expressions include:

- Using the wrong variable in an expression
- Loss of computational accuracy due to truncation
- Assigning a value to the wrong variable

Programming languages with "strong typing" rules, such as JOVIAL and Ada, do not provide implicit type conversions as does FORTRAN. Hence, more modern languages already enforce these rules.

[1]G. A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits of Our Capacity for Processing Information," Psychol. Review Vol. 63, pp. 81-97, 1956.

Data Flow Anomalies

Data flow analysis was originally developed as a technique for program optimization. Several ways were found to improve program efficiency by transformations based on the sequence of assignment and references to variables. Software quality researchers observed that the same conditions which led to optimization also indicated several kinds of program errors. Hence, data flow techniques have been incorporated into static analysis tools for program validation.

In the previous section on error detection (2.1.1), data flow and set/use analysis led to the detection of uninitialized variables. There are other patterns of assignment and references to variables which are considered anomalous, although they are not always erroneous. If a variable is set twice without any possible references to its first value, then the first assignment can be eliminated--or there is some error because of a missing reference. If the operation of the code is correct, then the first assignment can (and should) be removed to improve program efficiency and readability.

Another data flow anomaly is a variable which is set but never used afterwards. The final assignment could be eliminated if it were indeed not needed. This anomaly often occurs in program loops where a variable is updated at the end of the loop for the next iteration. When the loop terminates, the final value assigned may never be used. The warning message produced by data flow analysis will encourage the programmer to verify that the code is not in error. Sometimes, but not always, loops can be restructured to eliminate the anomaly.

Programs which have been repeatedly modified often contain declarations of variables and parameters that are no longer used. This anomaly reflects incomplete modification and is always a source of bewilderment when further modification is attempted. Often these extraneous variables (and particularly the unused parameters) indicate program errors. Data flow analysis can easily identify such anomalies because the list of set/use references will be empty for unused variables and parameters.

The following list summarizes the kinds of errors that can be detected by data flow analysis:

- Extraneous assignment statements
- Extraneous variables and parameters
- Missing variable and parameter references
- Statements out of sequence
- Uninitialized variables

Uninitialized variables and extraneous variables and parameters can always be detected. The other errors must be confirmed manually when anomalies are detected.

<u>Unreachable Code</u>

Program segments which cannot be reached from any other part of the program are called unreachable or "dead" code. Such code can never be executed and, hence, is not logically part of the program. This anomaly is much more likely to occur in "unstructured" programs, especially during modification. Programs with unreachable code may run correctly; therefore, unreachable code is not always considered a "hard" error. The presence of such code indicates either poor editing or a misconception on the programmer's part. When program storage space is at a premium, however, unreachable code should be classified as an error even if the program is otherwise correct.

Structural analysis techniques detect unreachable code by locating disconnected flow-graph components. One solution to the unreachable code problem is to automatically eliminate the offending program segments. Certain optimization techniques can sometimes result in unreachable code, and hence, optimizing compilers usually provide dead code elimination to minimize program storage requirements.

Another class of unreachable code is "logically" dead code which cannot be reached because the conditions for its execution can never occur. For example, in

```
if n>0 or n<10 then
   Statement-A
else
   Statement-B
endif
```

the condition (n>0 or n<10) is always true so Statement-A will always be selected and Statement-B is, therefore, logically unreachable. Techniques for identifying logically dead code are usually classified as formal techniques which are more powerful than those normally employed in static analysis.

Unreferenced Statement Labels

An unreferenced statement label is similar to a dead code anomaly, in that it is usually either caused by an oversight or is the result of a modification to the program. Unreferenced labels may not cause a program to run incorrectly, but they should be removed because they can confuse a reader of the code. Many compilers, as well as static tools, detect unreferenced statement labels.

Statement labels are a major source of errors in languages like FORTRAN IV that have no structured control constructs. In addition to unreferenced labels, FORTRAN programmers have problems with putting a label on the wrong statement, using the wrong label in a GOTO, or forgetting a label altogether. A compiler will catch the latter problem, since the label supplies information needed to generate the object code. The other errors cannot always be caught by static techniques. However, if a misused label causes another type of error to occur, then some of the other static error and anomaly checks discussed in this section may detect this condition. These include:

- Infinite loops
- Uninitialized variables
- Data flow anomalies
- Unreachable code

2.1.3 Assertion Checking

At least two kinds of program assertions can be checked by static analysis techniques. They are parameter and variable usage assertions and units assertions. Other assertions about the state of computations and the correctness of results are discussed in sections on dynamic and formal testing (Secs. 2.2, and 2.3).

Input/Output Assertions

Parameter and variable usage assertions are statements about how each parameter and variable is used within a program module. Parameters may be used for

- Input only
- Output only
- Modification (input and output)

Variables may be

- Local
- Global

Local variables must be declared in the local scope of the module and follow correct set/use rules. Global variables can be further classified by their input/output usage.

Input/output assertions provide a specification of the planned usage of parameters and global variables against which the actual usage can be checked. In correct programs this information is redundant since it could be derived from the code. However, during program development and in maintenance tasks these assertions quickly identify specification violations. Also, there is no run-time overhead associated with these assertions because all the analysis is done statically.

Some examples of program errors which can be detected by input/output assertion checking are:

- Inadvertent modification of input-only data
- Illegal use of the "old" value of output-only data
- Input-only or output-only use of data which is supposed to be modified

Violations in the first two of these categories are usually considered errors for reporting purposes. The third group contains many anomalies which are not always errors. Such anomalies indicate either program errors or incorrect assertions.

Units Assertions

In many engineering and physics problems it is standard practice to check consistency of the physical units used in the calculations as a check on the results. This technique can be used in programs as well by specifying the units for variables through assertions. Static checking for units violations is an extended form of type checking. Addition of real variables, for example, with different units such as feet and meters, becomes a mixed-mode operation which can be reported as an error. Units checking must also include knowledge of equivalent units such as VOLTS = AMPERES x OHMS to avoid extraneous warnings.

2.1.4 Test Data Generation

Generation of test data for dynamic testing is a good example of how static analysis can support other testing techniques. Several static analysis tools have been developed to aid in producing test data. These tools analyze COBOL source programs for file format information and can create test data files with randomly generated values. The created files serve two purposes: they can be used as sample data for exercising the program and they can be checked manually to see if they conform to program specifications and data entry forms. Both applications can provide valuable information.

Random data generation is usually not sufficient, however, for thorough dynamic testing and program validation. Programs must be tested on invalid as well as valid data to verify correct error processing. Also, special combinations of input data must be used to exercise a program thoroughly. Random test data is usually not effective in these cases but other techniques such as symbolic execution can provide the needed testing support.

2.1.5 Program Documentation

A very important product of static analysis, in addition to error and warning messages, is program documentation. Documentation reports do not explicitly indicate program errors but it is often possible to detect additional errors manually by studying these reports. For program maintenance and modification, static analysis documentation is of tremendous value. Below is a list of some of the kinds of documentation which can be produced from information collected by static analysis tools.

- Global cross-reference report indicating input/output usage for variables in all modules.
- Module invocations report indicating the calling modules and showing all calling statements.
- Module interconnection report showing the program's module calling structure.
- Special global data reports for variables in COMMON blocks and COMPOOLS.
- Program statistics including total size, number of modules,

module size distribution, statement type distribution, and complexity measures.

- Summaries of analyses performed, program statistics, and errors and warnings reported.

The interested reader will find a variety of example documentation reports in the user's manuals for tools such as FAVS,[1] JAVS,[2] and CAVS.[3]

2.2 DYNAMIC TESTING TECHNIQUES

Dynamic testing covers all program testing techniques which involve running compiled code and observing the output produced. These techniques are divided into the following categories for our discussion:

- Program testing facilities - intended primarily for detecting the presence of program errors
- Program debugging facilities - intended primarily for locating errors once they are detected
- Program performance measurement

In this section we discuss tools and techniques for both detecting program errors (testing) and locating them for correction (debugging). Many techniques classified under "testing" are equally helpful in program debugging. Those listed under "debugging," however, are designed to isolate error sources and are considered less useful for verifying that large programs operate correctly.

---

[1]D.M. Andrews and R.A. Melton, FORTRAN Automated Verification System User's Manual, General Research Corporation CR-1-754/1 (April 1980).

[2]C. Gannon and N.B. Brooks, JOVIAL J73 Automated Verification System Functional Description, General Research Corporation CR-1-947 (March 1980).

[3]M. Sharp, R. Melton, and G. Greenburg, COBOL Automated Verification System Functional Description, General Research Corporation CR-2-970 (November 1980).

Most of the techniques discussed here are batch oriented for use in testing software destined for production applications. The only interactive testing facilities found were "test harness" programs which allow a user to control test parameters. A number of interactive debugging tools have been developed for popular high-level programming languages. However, no testing tools (as opposed to debugging tools) were found for languages designed primarily for interactive use.

2.2.1 Program Testing Facilities

Techniques intended primarily for detecting the presence of program errors can be further grouped into the following categories:

- Executable assertions
- Structural testing, supported by instrumentation tools
- Functional testing, supported by test harness techniques

Structural testing includes several methods of evaluating testing thoroughness by different measures of test coverage. Test harness techniques are used to exercise individual program modules and sub systems with more rigorous tests than can usually be applied in system-level tests. Each of these categories is elaborated on below.

Executable Assertions[1]

An assertion is a logical statement about the state of a computation within a program. Formal verification techniques use assertions to prove properties of programs. For dynamic testing, it is possible to evaluate assertion statements during execution and check to see that the specified conditions hold. Assertion violations are typically reported on the standard output media where they are difficult to ignore. The basis of testing with assertions is trying to force the program into violating the conditions expressed in the assertion statements.

[1] J. P. Benson and S.H. Saib, "A Software Quality Assurance Experiment," Software Quality and Assurance Workshop, pp. 87-91, San Diego, Nov. 15-17, 1978.

Most common programming languages do not include assertions as a statement type. (Ada does, but COBOL, FORTRAN, JOVIAL, PL/I, and Pascal do not.) However, it is relatively easy to translate an assertion into an "if" statement to test the asserted condition, using a preprocessor or other translation technique. Many structured programming preprocessors and several compilers provide assertion statement translations.

Assertion testing requires the insertion of meaningful assertion statements in the program to be tested. This can be an onerous task if the program is large and not well structured. If the assertions are inserted during program development, however, several advantages are gained: the task is distributed over time; the assertions often reiterate program specifications and hence can be cross-checked manually; module and subsystem tests can include assertion testing; and a very stylized form of internal program documentation, formed by the assertions, is kept up-to-date for testing purposes.

Structural Testing

The purpose of structural testing is to ensure that the test program has been explored thoroughly. Several measures of the thoroughness of structural testing can be used:

- Statement coverage
- Decision-to-decision path or branch coverage
- Linear code sequence coverage
- Dynamic data flow analysis

The simplest measure of testing thoroughness is the percentage of statements executed during a test run. This test coverage statistic can be used as a testing criterion by setting a goal such as "95 percent statement coverage" for a series of tests. Instrumentation tools typically report the number and percentage of statements covered and identify the statements that were missed. Current tools have not yet incorporated automatic test case generation to maximize coverage metrics. While this may be a possibility for the future, selecting data to exercise particular statements is a manual task at present.

Unfortunately, the statement coverage technique has some pitfalls and several kinds of program errors may go undetected even with 100 percent coverage. Two obvious tests which can be easily missed are testing both the "true" and the "false" branches of an "if" statement which has no "else" clause, and testing a "while" loop where the iteration condition is initially "false" so that the loop body is not executed. The first test could indicate a missing "else" clause, and the second, an incorrect loop termination condition or improper loop initialization.

Two techniques have been developed to overcome some of these problems using program units called "decision-to-decision paths" (DD-paths) and "linear code sequence and jump" (LCSAJ's).[1] Both of these techniques provide improved testing based on coverage metrics. DD-path instrumentation records the path taken at each branch point (based on the program's flow graph) and at procedure and function entry and return points. LCSAJ's are based on a program's source text rather than on its flow graph. Each contiguous sequence of executed source statements forms an LCSAJ. Hence, an LCSAJ can be described by its starting and ending line numbers, and the number of the line where execution proceeds after the break.

Howden has shown that decision-to-decision path or "branch" coverage testing is more effective than statement coverage.[2] The experiment conducted by Woodward et al. indicates that LCSAJ coverage is usually lower than DD-path coverage for a given test. This means

[1]M. Woodward, D. Hedley, and M. Hennell, "Experience with Path Analysis and Testing of Programs," IEEE Transactions on Software Engineering, Vol. SE-6, No. 3 (May 1980), pp. 278-286.

[2]William Howden, "Theoretical and Empirical Studies of Program Testing," IEEE Transactions on Software Engineering, Vol. SE 4, No. 4 (July 1978), pp. 293-298.

that a more rigorous test is usually required to attain a given level of coverage using the LCSAJ measure. Although this is not always true for incomplete tests, complete LCSAJ coverage subsumes complete DD-path coverage.

Woodward also developed a generalized coverage measure based on multiple LCSAJ sequences. The first level in this hierarchy of testing metrics is the basic coverage measure. The next level requires, in addition, testing all possible pairs of successive LCSAJ's. In general, the N<u>th</u> level in this scheme requires testing all possible sequences of N or fewer consecutive LCSAJ's. The same strategy can be applied to sequences of DD-paths to improve testing coverage. Pairwise DD-path coverage effectively tests all possible compositions of operations performed between decision points, and is, intuitively, a more complete test than simple branch coverage.

There are two areas of difficulty in the current state of DD-path and LCSAJ testing methodology. One is the distortion in the metrics caused by infeasible sequences of program code. Coverage statistics are currently based on all possible structural or textual sequences, a measure which does not account for logically impossible combinations. Hence a rating of 100 percent coverage may not be attainable even though all feasible sequences of code are tested. Identification of infeasible code sequences would make these measures more accurately reflect the degree of testing achieved. Infeasible paths, however, are often not amenable to completely automated detection because of program size. Semi-automated techniques using symbolic execution, for example, may prove to be the most effective way to determine if code sequences which have not been exercised are indeed logically impossible.

The second difficulty is that of generating test data to maximize coverage measures. Test coverage reports typically indicate the code segments missed in a test run. However, determining the data values required to traverse a particular sequence of statements in a large program can be a complex task. Branching conditions based on non-linear transformations of the input data seem to cause the most difficulty. Symbolic execution and formal verification techniques could aid this task in an integrated testing and verification facility.

The general strategy for all structural techniques is to add software "probes" to a program under test to record information about the program's execution behavior. Instrumentation tools can insert probes automatically. Running an instrumented program produces a trace file containing information from each test probe encountered. After a test run, the trace file is read by an analysis program to condense the data and report test results. The differences between the various instrumentation tools stem from the kinds of information extracted by the test probes and the analyses of the trace file data.

A method for detecting data flow anomalies dynamically is described by Huang.[1] Test probes record all set and use references to program variables during execution. The trace file is then analyzed for anomalous data references. This approach can detect anomalous array references which cannot be detected by static data flow analysis.

Functional Testing and Test Harness Techniques

Functional testing involves the creation of test data based on program requirements specifications followed by the verification that the output produced meets the requirements.[2] This technique is often used by software buyers in preparing acceptance tests for new software.

[1] J. C. Huang, "Detection of Data Flow Anomaly Through Program Instrumentation," IEEE Transactions on Software Engineering, Vol. SE-5, No. 3 (May 1979), pp. 226-235.

[2] William Howden, "Functional Program Testing," IEEE Transactions on Software Engineering, Vol. SE-6, No. 2, pp. 162-169 (March 1980).

Clear, well written, and testable program requirements obviously contribute significantly to the quality of testing that can be accomplished by this method. These tests, however, can be created early in the software development cycle and can serve as additional detailed specifications for program designers and implementers.

A software test harness is a program which provides an environment for testing individual software modules as well as complete programs.[1] Typically, it includes:

- Acting as a substitute "main program" for the software under test
- Filling in for missing software components
- Controlling the execution of a test
- Monitoring a test's progress

The facilities provided by a software test harness correspond to the test and measurement equipment on an electronic technician's workbench. Some of the advanced features found in test harness tools include:

- Automatic test control
- Automatic test data generation
- Interactive test control
- Test history accounting
- Verification of computed results

Unfortunately, no single test harness currently provides all of these capabilities.

[1] Frederick J. Drasch, and Richard A. Bowen, "IDBUG: A Tool for Program Development," Proc. Software Quality Assurance Workshop, San Diego, November 15-17, 1978, pp. 106-110.

Most test harnesses are designed as debugging tools for locating known errors and provide low-level testing controls. The techniques, however, could also be applied equally well to testing software at a higher level and verifying its correct operation. For example, debugging harnesses typically do not keep track of path coverage. Their low-level control, however, is ideal for special values testing. The combination of these facilities with a well engineered human interface would make a very effective test tool.

2.2.2 Program Debugging Facilities

The emphasis of the testing techniques discussed above is on detecting the existence of program errors. Locating errors for correction is a different task with different requirements for automated support. Several of the testing techniques already described also provide useful debugging information. All of the static analysis techniques, for example, can provide excellent error-locating diagnostics, often indicating the exact source of an error. Using executable assertions is one of the best ways to produce good error diagnostics during program execution. Most of the other dynamic techniques produce no error diagnostics per se. Path coverage information, however, can be very helpful in reducing the scope of a search for program bugs.

Executable assertions provide excellent debugging information when the asserted conditions are not too complex. Assertion violation reports typically refer to the source code line number where the assertion appears and are interspersed with the normal program output. Hence, the programmer is able to determine both the location of the violated condition and the context of the error. Keeping assertions simple maximizes the amount of information reported. That is, the statements

```
ASSERT (N>0)
ASSERT (N<=100)
```

provide better diagnostic information than

ASSERT (N>0 and N<=100)

which states the same conditions.

If a relatively simple test case can be constructed to illustrate a program error, then path coverage reports can help reduce the scope of the search for the error. These reports indicate which paths have been exercised by the test data that expose the error. The simplest such test case, therefore, traverses the fewest number of extraneous (i.e., correct) paths and focuses the programmer's attention on relevant parts of the program source code.

Additional tools have been developed to help locate errors once they have been detected. These include several kinds of trace facilities and dump formatters. A tracing tool produces a continuous log of the operations being monitored such as statements, procedure calls, or assignment to individual variables. Program errors typically appear as incorrect sequences of operations in trace logs. Identifying missing, extra, or out-of-order operations will usually lead a programmer to the error. One drawback of this technique is that sequence anomalies may be as difficult to find in very long trace logs as the errors themselves. Some tracing tools allow the user to turn the trace log on and off dynamically to reduce the amount of output generated.[1]

Modern program dump facilities provide useful information for locating errors which cause abnormal program termination. The immediate cause of an aborted program execution, the source code module name and line number, and the sequence of procedure invocations at the time of the fatal operation are reported. In addition, the values of program

[1]R. E. Griswold, J. F. Poage, and I. P. Polousky, The SNOBOL4 Programming Language (2nd ed.), Prentice-Hall, 1971, Chapter 8.

variables may be reported. The actual error(s) which precipitated the abnormal termination can usually be determined from this information. Octal or hexadecimal core dumps are not considered tools for debugging. However, there are commercially available tools which attempt to interpret raw core dumps and produce reasonable diagnostic reports.[1]

2.2.3 Program Performance Measurement

A classical method for measuring program performance using dynamic testing techniques is to record the elapsed CPU time at the entry to and exit from every procedure invocation. This data is then condensed and reported showing the total and average times spent in each program module. Most computer operating systems provide procedures for obtaining the necessary timing information, although there is no standard method.

For program segments with extreme timing requirements, execution time can be determined statically by summing the execution times of the individual machine instructions which will be executed. These analyses typically assume that no delays or interrupts will interfere with the normal sequence of operations. Of course, instruction-set timing figures are unique to each make and model of computer, so these are not very general-purpose tools.

2.3 SYMBOLIC EXECUTION AND FORMAL VERIFICATION

Two of the largest areas of current active research in software testing techniques are symbolic execution and formal program verification. Symbolic execution is a method of interpreting programs by deriving mathematical expressions for the values of variables rather than actually computing their numerical values. The expressions produced show a perspective of the progress of computations which is very different from other means of testing such as tracing intermediate

[1] Datapro Directory of Software, Datapro Research Corporation, August 1980.

values. The additional information obtained from symbolic execution has been shown to improve detection of several kinds of program errors.[1]

Formal verification is a more rigorous approach to software testing which involves proving properties of computations performed by programs. This technique provides probably the highest degree of assurance of program correctness but is also the most difficult to apply. Two prerequisites for formal verification of a program are precise specifications of the inputs and required outputs for the computation, and formal specifications of the semantics of the programming language used. Program correctness is proven by showing that, given the specified input conditions and the rules of the programming language, execution of the program will terminate and produce the desired output conditions.

Automated tools which support these techniques are discussed in the following sections. All of the tools which have been implemented recognize either specially designed mini-languages or restricted subsets of full-fledged languages. They are typically vehicles for research in program testing and verification, and are not commercially available software products.

Symbolic Execution

Symbolic execution has been described by King[2] as an intermediate technique between dynamic testing and formal program verification. Instead of executing a program with test input data, the inputs are represented "symbolically" and the output produced is in the form of

[1] William E. Howden, "Symbolic Testing and the Dissect Symbolic Evaluation System," IEEE Transactions on Software Engineering, Vol. SE-3, No. 4 (July 1977), pp. 266-278.

[2] James C. King, "Symbolic Execution and Program Testing," CACM, Vol. 19, No. 7 (July 1976), pp. 385-394.

mathematical expressions rather than computed results. For example, a loop to sum the elements of an array

```
SUM = 0.0
DO (I = 1,4)
    SUM = SUM + A(I)
END DO
```

when symbolically executed would yield the expression

```
SUM = A(1) + A(2) + A(3) + A(4)
```

Such results are readily checked formally or informally for correctness.

The symbolic execution of an example program with errors, used by Howden,[1] produced

$$SIN(X) = X + X^3/6.0 + X^5/120.0$$

for the approximate value of the sine function. The errors are easily spotted by comparing this result with the expected solution,

$$SIN(X) = X - X^3/6.0 + X^5/120.0 - X^7/5040.0$$

The results of dynamic testing of this program would probably have indicated some lack of accuracy in the result. The symbolic execution, however, for this example, provides considerably more information about the nature of the errors.

The two major components of a symbolic execution tool are a program interpreter and an expression simplifier. The interpreter is responsible for recognizing program statements and translating their execution into algebraic expressions. This task is very similar to

[1]Howden, op. cit.

compiling or interpreting a program and computing actual results rather than symbolic results.

Expression simplification is the key to making symbolic execution palatable to human users. The expressions produced by the interpreter can become quite complicated in just a few steps of computation. For example, the "raw" form of the expression for the correct sine function above is

```
SIN(X) = ((X + X * (-X*X/(3.0 * 2.0)))
              + X * (-X*X/(3.0 * 2.0)) * (-X*X/(5.0 * 4.0)))
              + X * (-X*X/(3.0 * 2.0)) * (-X*X/(5.0 * 4.0))
                  * (-X*X/(7.0 * 6.0))
```

which is not easy to identify as a correct solution. Considerable knowledge of algebraic transformations and simplifications must be built into the interpreter to produce readable output.

Symbolic execution for program testing has been uniformly presented as an interactive technique. Users select symbolic values to be displayed and control program interpretation at a very low level. This approach has been successful for testing small example programs and segments of larger programs. However, this degree of detail limits the effectiveness of the technique in testing large scale programs.

<u>Formal Verification</u>

A formal verification of a program's correctness is a proof of the following theorem:

> If all of the initial conditions hold at the start of execution, then, when the program terminates, the final conditions will be satisfied.

Hence, both the initial and final conditions must be defined in sufficient detail and precision so that mathematical proof techniques can be applied.[1] Also, the semantics of the programming language must be defined with sufficient rigor to allow logical reasoning about each step of the computation.[2]

Tools which have been developed to support formal program verification can be grouped into three categories:

- Proof generators
- Proof verifiers
- Verification condition generators

These are listed in decreasing order of computational complexity.

A proof generator will take a program along with its initial and final assertions and derive the theorem by a sequence of logical operations. Of course, if the program is not correct then the theorem cannot be proved and this process will fail. Proof generators are complex programs which can interpret the assertions and program statements, apply complex rules of inference and various heuristic proof strategies, and eventually determine if the input program satisfies the assertions.

Proof verifiers solve the simpler (by comparison) problem of checking a manually generated proof. Additional assertions must be provided manually along with any lemmas which the proof verifier will need to validate each step from assertion to assertion. The proof verifier need only interpret sequential program statements and be able

[1]Robert W. Floyd, "Assigning Meanings to Programs," Proc. Symp. in Applied Mathematics, Vol. 19, American Math. Soc., Provincetown, R.I. (1967), pp. 19-21.

[2]C.A.R. Hoare and Niklaus Wirth, "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica, Vol. 2 (1973), pp. 335-355.

to apply simple rules of inference. An advantage of this approach is that a program error can be isolated to a small segment of code for which the initial and final assertions are fully specified.

Verification condition generators typically use symbolic execution techniques and work "backward" from the final assertion to derive the necessary pre-conditions for each statement in the program. If the initial assertion implies the pre-conditions for the first statement then the program is correct. Verification condition generators, however, do not provide the facilities to determine whether this implication holds. Although some systems do attempt to simplify the expressions generated, the greatest difficulty with this approach is the complexity of the automatically generated assertions.

PRECEDING PAGE BLANK-NOT FILMED

3 SOFTWARE TESTING METRICS

It is important to examine testing in the context of the overall software development process. A comprehensive evaluation of the software testing techniques described in Chapter 2 should determine their ability to enhance software quality, and weigh this against the costs in time, manpower, and computer resources of using them.

Unfortunately, the current state of knowledge precludes a detailed "cost-benefit" approach to evaluating test techniques. This report can make no comparisons of the "amount" of quality enhancement provided by the various test techniques. Instead, we develop various effectiveness and cost criteria and rate the individual techniques on these.

3.1 THE GOALS AND PROBLEMS OF DEVELOPING METRICS

In any science, measurements are tools which help in making judgments about the behavior of a system. When software is being tested, many kinds of judgments must be made. For instance:

- What is the nature of the errors in the software -- how many are there, what kinds, where are they?
- What methods can be used to find the errors effectively and inexpensively?
- How can one be sure that the software works correctly at any point in time?

It is very desirable to have measures which provide a quantitative basis for making such judgments. However, many aspects of software testing cannot be quantified. This section discusses how testing measures can be useful to managers of software development projects. It also considers the problems involved in making such measurements.

3.1.1 Management Concerns in Testing Software

Advanced software testing techniques are being used by industry and government, but practices vary widely. A survey of 60 software development projects in the US aerospace industry[1] concluded that only 12% of the projects surveyed had formal standards on how to plan for tests, and only 41% produced formal quality assurance plans.

Consideration of the full software life-cycle is critical to proper management of software development in general and testing in particular. Testing must be formally incorporated into the software development plans from the beginning of a project, and procedures for configuration management that institute control of change and integration must be in place early. The great danger in haphazard management and lack of planning is that "by the end of the validation/verification phase (at installation time), corporate level management cannot do a great deal more to influence the quality of the product."[2] If things are going wrong with a project, management must find out and take corrective action as early as possible.

The overall quality of a software product is composed of many factors. Testing should improve the quality of a software project in each of the areas discussed below.

Correctness may be thought of as the lack of errors in a program. Programs, particularly large ones, usually have errors, so sometimes it is desirable to try to judge "how correct or incorrect" a program is. An estimate of the number of errors remaining in the program is one

[1] R. Thayer, A. Pyster, and R. Wood, Results of a Survey on Management Techniques and Procedures used in Software Development Projects by the U.S. Aerospace Industry, SM-ALC/MME TR 79-54, Volume II.

[2] J. S. Cooper, "Corporate Level Software Management", IEEE Transactions on Software Engineering, Vol. SE-4, No. 4 (July 1978), p. 324.

"degree of correctness" metric. Another metric that has been used is an estimate of the program's reliability. Here reliability has a meaning similar to the use of the term in the hardware context--how often will the program fail? These measures are considered in greater detail in Sec. 3.2 .

Compliance with specifications - whether the program does everything that it is supposed to do, in the way that it is supposed to. To simplify matters, we discuss testing with the assumption that specifications are given and fixed. We also assume that a functional specification is available which can be used to determine if the program is operating correctly. Traditionally, testing detects errors only if the errors are reflected in the program's output behavior. However, certain test tools are useful for assuring compliance with other specifications - for instance, instrumentation tools can help to increase the speed of execution of a program.

Cost may not normally be thought of as a measure of software quality, but the two are intimately connected in the government and corporate environment. Testing is an important link between quality and cost since it is a major determining factor of both. In Sec. 3.3 we examine the effect of testing on total software development costs.

Other desirable characteristics - there are some standard software quality characteristics which should be present in a finished software product regardless of whether they are covered in the product's formal specifications. Examples of such characteristics are code readability and ease of use. Boehm, et al.,[1] have constructed a "Software Quality

[1] B. W. Boehm, J. R. Brown, M. Lipow, "Quantitative Evaluation of Software Quality", Proceedings - 2nd International Conference on Software Engineering, San Francisco, October 13-15, 1976, pp. 592-605.

Characteristics Tree" (whose elements are commonly known as "-ility" measures) which covers many of these items. Such qualities tend to be very difficult to measure quantitatively. In Sec. 6.5 we look at how the test techniques can contribute to assuring that these qualities are present in software.

3.1.2 Making Measurements of Software

If we impose the same standards of rigor on software measurements that are required of measurements in the physical sciences, the following must exist for a software metric to be properly defined:

- A principle of operation known to hold for all applicable software
- A quantity with well-defined units to be measured
- An accurate means of performing the measurement
- A means of directly relating the quantity measured to a software quality or test effectiveness characteristic

Unfortunately, software metrics typically fail to satisfy at least one of the above criteria. Perhaps the most difficult of these criteria is the first, since many important rules about the behavior of software have not been firmly established. It may well be that the field of software testing, since it deals with man-made objects and systems rather than those developed by nature, must be content with the status of "inexact science", which is sometimes accorded to the social sciences. If this is the case, the scientific methods of measurement and experimentation, while still quite valid activities in the discipline, should be expected to lead only to heuristic approaches or "rules-of-thumb" for problem solving rather than to exact formulas.

For most software quality concepts, quantitative measures alone do not provide a complete basis for evaluation. A good example of this is the problem of determining the adequacy of the documentation of a piece of software. A metric cannot measure the relative importance of the documentation reports produced against those not produced, or evaluate how easy the produced documentation is to understand and use. The process of evaluating software quality and test effectiveness must proceed under the awareness that qualitative evaluations should accompany the use of metrics.

None of the above criticisms of software metrics should be interpreted as denying the fact that good metrics can provide valuable management and technical information. Five areas in which metrics can make contributions to testing and software development are listed below.

- Metrics can guide the testing process by identifying what needs to be tested and indicating what test techniques might be most useful.
- Metrics provide a means of recording project status.
- Metrics can provide parameters for cost estimation.
- Metrics can provide aids to software maintenance, both as a form of documentation of the development of the system and as predictors of the location and severity of possible problem areas.
- Metrics may be used in formulating requirements for systems that are expected to fulfill some measurable criteria. Examples of such requirements include minimum standards of test coverage or operational reliability.

3.1.3 Practical Problems with Software Quality and Testing Metrics

Although software metrics have received a great deal of study, there is still a need for experience and data on their use in "live" development projects. Data has been gleaned from experiments with small, well-understood programs and post-facto analyses of completed projects, but there are some obstacles to the collection of data from ongoing projects. These include:

- Data collection requires extra record keeping during the testing stage, which is already burdened with data and details.
- Persons responsible for the quality of a project (from programmers on up) may feel that the data will be used to form unfair evaluations of their work.
- It is very difficult during "live" conditions to control the elements of human ability and judgment that affect both data collection and the outcome of the project itself.

Test techniques that require user intervention and judgment are very difficult to rate in terms of effectiveness because human factors need to be taken into consideration. This report does not address the problem of measuring the skill with which a (human) tester applies test techniques, although we do examine the effort and level of expertise required to apply the techniques.

3.2 MEASURES OF TEST TECHNIQUE EFFECTIVENESS

The metrics that we consider in this section can help to make the following judgments about the effectiveness of a test technique:

- What kinds of errors will the technique detect?
- How many of the errors in a program can be detected by using the technique?
- How completely has the technique been applied at any point in the testing process? Is it possible to apply the technique exhaustively, or is the number of tests that can be made essentially infinite?
- When can testing stop?

3.2.1 Types of Errors Detected

Enumerating error types requires the use of an error classification system. We chose to use the one developed at TRW[1], partly because it had been used in previous GRC studies[2,3] of test techniques. The error categories in this scheme are listed in Table 3.1. We had hoped to use the TRW error classification to develop a chart which listed the error types detected by each of the test techniques. However, we later decided that this approach would not be valid. The main problem with this classification scheme is that it emphasizes instances of errors, while testing is often concerned with only the symptoms of errors.

Errors in computer programs are caused by faulty human actions. But it is often convenient to classify errors at the level of the computer instead of putting them in human terms. To do this, we need to make a distinction between an error in a program's source code and the effect that it has on program operation. The source code problem can be termed the "instance" of the error, while the run-time problem is the "symptom".

[1] T. A. Thayer et. al., Software Reliability Study, Rome Air Development Center RADC-TR-76-238, August 1976, pp. 3-18 to 3-20. This is the "Project 5" error classification. This document is hereafter referred to as "the TRW report". The report has recently been published by North-Holland, New York.

[2] C. Gannon, R. N. Meeson, and N. B. Brooks, An Experimental Evaluation of Software Testing, General Research Corporation CR-1-854, May 1979.

[3] J. P. Benson and D. M. Andrews, Adaptive Search Techniques Applied to Software Testing, General Research Corporation CR-1-925, February 1980.

TABLE 3.1
THE TRW "PROJECT 5" ERROR CLASSIFICATION SYSTEM
(Source: Thayer, et al., pp. 3-18 to 3-20)

A_000	**COMPUTATION ERRORS**
A_100	Incorrect operand in equation
A_200	Incorrect use of parenthesis
A_300	Sign convention error
A_400	Units or data conversion error
A_500	Computation produces an over/under flow
A_600	Incorrect/inaccurate equation used
A_700	Precision loss due to mixed mode
A_800	Missing computation
A_900	Rounding or truncation error
B_000	**LOGIC ERRORS**
B_100	Incorrect operand in logical expression
B_200	Logic activities out of sequence
B_300	Wrong variable being checked
B_400	Missing logic or condition tests
B_500	Too many/few statements in loop
B_600	Loop iterated incorrect number of times (including endless loop)
B_700	Duplicate logic
C_000	**DATA INPUT ERRORS**
C_100	Invalid input read from correct data file
C_200	Input read from incorrect data file
C_300	Incorrect input format
C_400	Incorrect format statement referenced
C_500	End of file encountered prematurely
C_600	End of file missing
D_000	**DATA HANDLING ERRORS**
D_050	Data file not rewound before reading
D_100	Data initialization not done
D_200	Data initialization done improperly
D_300	Variable used as a flag or index not set properly
D_400	Variable referred to by the wrong name
D_500	Bit manipulation done incorrectly
D_600	Incorrect variable type
D_700	Data packing/unpacking error
D_800	Sort error
D_900	Subscripting error
E_000	**DATA OUTPUT ERRORS**
E_100	Data written on wrong file
E_200	Data written according to the wrong format statement
E_300	Data written in wrong format
E_400	Data written with wrong carraige control
E_500	Incomplete or missing output
E_600	Output field size too small
E_700	Line count or page eject problem
E_800	Output garbled or misleading

TABLE 3.1 (Concl.)

F_000	INTERFACE ERRORS
F_100	Wrong subroutine called
F_200	Call to subroutine not made or made in wrong place
F_300	Subroutine arguments not consistent in type, units, order, etc.
F_400	Subroutine called is nonexistent
F_500	Software/data base interface error
F_600	Software user interface error
F_700	Software/software interface error
G_000	DATA DEFINITION ERRORS
G_100	Data not properly defined/dimensioned
G_200	Data referenced out of bounds
G_300	Data being referenced at incorrect location
G_400	Data pointers not incremented properly
H_000	DATA BASE ERRORS
H_100	Data not initialized in data base
H_200	Data initialized to incorrect value
H_300	Data units are incorrect
I_000	OPERATION ERRORS
I_100	Operating system error (vendor supplied)
I_200	Hardware error
I_300	Operator error
I_400	Test execution error
I_500	User misunderstanding/error
I_600	Configuration control error
J_000	OTHER
J_100	Time limit exceeded
J_200	Core storage limit exceeded
J_300	Output line limit exceeded
J_400	Compilation error
J_500	Code or design inefficiency/not necessary
J_600	User/programmer requested enhancement
J_700	Design nonresponsive to requirements
J_800	Code delivery or redelivery
J_900	Software not compatible with project standards
K_000	DOCUMENTATION ERRORS
K_100	User manual
K_200	Interface specification
K_300	Design specification
K_400	Requirements specification
K_500	Test documentation
X0000	PROBLEM REPORT REJECTION
X0001	No problem
X0002	Void/withdrawn
X0003	Out of scope — not part of approved design
X0004	Duplicates another problem report
X0005	Deferred

The following example illustrates the distinction between the cause, instance, and symptom of an error.

```
REAL FUNCTION AREA(R)
DATA P /3.14159/
AREA = PI * R**2
RETURN
END
```

Here the DATA statement has been typed incorrectly - "P" should be "PI". This is the error at the human level, or the "cause" of the problem. However, at the source code level, the program contains an uninitialized variable, "PI", and one that is set and not used, "P". This is the instance of the error. To a user of the routine, the problem is that the area is computed incorrectly; this is the symptom of the error.

For most of the TRW categories, it is very difficult to decide whether to give a technique credit for being able to detect that type of error. For example, a few error categories correspond exactly to certain static error and anomaly checks (e.g. "D_100 Data initialization not done" with the static "uninitialized variable" error); but other categories (e.g. "D_400 Variable referred to by the wrong name") describe errors that static analysis may or may not catch. Similarly, only a few of the error categories describe aspects of program behavior that are diagnosed during dynamic testing.

The misspelling error presented above might be classified in any of the following ways under the TRW scheme:

- A_100 Incorrect operand in equation
- A_800 Missing computation ("PI=P" would correct the problem)

- D_100 Data initialization not done
- D_200 Data initialization done improperly
- D_400 Variable referred to by wrong name

There are no detailed, written descriptions of the TRW error categories, so which of these five should actually be chosen depends entirely upon the interpretation and preferences of the individual doing the classifying.

Now consider how this same error looks to each of the test techniques which we are evaluating:

- Static analysis will report two data flow violations: "P is set but not used", and "PI is used but not set".

- Dynamic testing must catch the fact that the value returned by the function is wrong.

- Formal testing will find the error when it cannot be established that "PI" has a value close to what is desired.

The only close correspondence between a TRW error category for this error and a test technique detection method is the category "D_100" and the "PI used but not set" message from static analysis. None of the error categories suggest the ways that the dynamic or formal techniques would detect the error.

In Sec. 4.1 we present a chart which shows the major TRW error categories that are addressed by each test technique. The chart is based on experience with the test techniques in error seeding experiments and real projects. The relationship between the techniques and the error categories is empirical, not analytical. As we note in Sec. 4.2, only in a very few cases is there a high degree of certainty that a test technique will detect a particular TRW error type.

3.2.2 The Percentage of Program Errors Detected

A natural question to ask about a test technique is how many of the errors in a program can be detected by using the technique. This can be expressed with a metric as the ratio:

$$\frac{\text{Number of errors detected}}{\text{Total number of errors in a program}}$$

Studies which have produced error detection ratio data for the test techniques have been of four types:

- Case studies of on-going development projects
- Analysis of historical data from real projects
- Experiments using "artificial" programs and/or errors
- Theoretical analyses in which test techniques were not actually applied

Of these methods, the most scientific are the case studies; unfortunately, only a few have been performed. Historical data is less acceptable because often it does not link the detection of an error explicitly to the use of a test technique. Experiments cannot duplicate the process of developing a major piece of software. Not enough is known to "analytically" determine error detection ratios for the test techniques without actually applying them.

Studies which have provided values for the error detection ratio metric are cited in the profiles of each test technique in Chapter 5. The results were obtained under widely varying conditions, and represent initial attempts at determining representative data for the metric. The data currently available is not sufficient to make conclusive quantitative comparisons between different test techniques on the basis of this metric.

3.2.3 Completeness-of-Testing Metrics

The general form of a completeness metric is the ratio:

$$\frac{\text{Number of tests performed}}{\text{Total number of tests possible}}$$

The completeness metric for each of the test techniques is given in Table 3.2. The metric takes a specific form for each test technique. For some of the techniques, 100% completeness is achievable, while for others it is a practical impossibility. Lower and upper bounds on the completeness ratio—representing minimum acceptable and maximum feasible levels of testing for each technique--would be very useful to have. Unfortunately, such bounds have not been determined for those techniques for which 100% completeness cannot be achieved.

The completeness metric for static analysis assumes that the tool or technique used makes a fixed number of error and anomaly checks whenever it is used. When comparing the effectiveness of static testing when different tools are used, one must consider the different types of checking that are done. In general, errors found by static analysis are violations of the semantics of the programming language. Therefore, a metric for rating static analysis tools is:

$$\frac{\text{Number of semantic error types checked}}{\text{Total number of semantic errors}}$$

The problem with the completeness metric given for executable assertion testing is that it is hard to estimate the number of assertions that are required. In Sec. A.5.3 of Appendix A (Guidelines for Testing Software) we present a list of locations where assertions should appear in a program. The completeness metric for assertion testing should be calibrated to the number of these locations that exist in a program.

TABLE 3.2

COMPLETENESS METRICS FOR THE TEST TECHNIQUES

Technique	Metric	100% Complete Testing Possible?
Static Analysis	$\frac{\text{Number of modules checked}}{\text{Total number of modules}}$	Yes
Executable Assertions	$\frac{\text{Number of assertions used}}{\text{Number of assertions required}}$	Yes
Functional Testing	$\frac{\text{Number of input tests made}}{\text{Number of input tests required}}$	No
Instrumentation/Structural Testing	$\frac{\text{Number of structural units tested}}{\text{Total number of structural units}}$	Yes: statements, branches No: LCSAJ's, paths
Symbolic Execution	$\frac{\text{Number of symbolic expressions derived}}{\text{Total number of output variables}}$	Yes: modules, small programs No: large or complex programs
Formal Verification	$\frac{\text{Number of program assertions proven}}{\text{Total number of assertions}}$	Yes: certain properties of programs No: general correctness of most large programs

Several different structural units can be used in the completeness metric for structural testing. These include statements, branches, combinations of branches, and paths. Complete branch coverage, which subsumes *statement* coverage, is possible for almost all programs. It is usually impossible to test all paths in a program, or even all possible combinations of N or fewer branches if N is 3 or more.

Functional testing is the most "open-ended" of the test techniques considered—the number of possible tests is always very large, and so the completeness metric is not very helpful. For example, testing all input combinations for a program with six input variables, each of which may take on three values, requires 729 test cases. The completeness metric gives no information about the relative importance of the tests that are run and not run.

The limitations in applying formal *techniques are determined* by the ability of the test tool or the user to simplify complicated symbolic expressions and to provide information necessary to proceed with the formal reduction of the program. An incomplete symbolic execution is one in which there is some output variable for which a symbolic expression in terms of input variables and constants has not been derived. An incomplete formal verification exercise is one in which the truth or falsehood of an assertion about program behavior has not been established.

3.2.4 <u>How Long Should Testing Continue?</u>

For the techniques of functional testing, executable assertions, and path testing, 100% complete (exhaustive) testing may be a practical impossibility. Under these circumstances, some other criterion for ending the testing process must be used. We present three possible criteria here:

- A "marginal benefit" metric, which indicates when the value of additional tests becomes negligible.

- Techniques for estimating errors--the number of errors actually found can be compared to this number to indicate whether testing has been thorough enough.

- Reliability models, which attempt to predict the distribution of failures during the operation of the program.

The "marginal benefit" of running an additional test case is the number of errors found by running that test. Paige and Balkovich[1] postulated the relationship shown in Fig. 3.1 between the number of tests run and the number of errors found. The basis for their hypothesis is that initial tests sift out a large set of errors that are more easily detected, and that a second peak in the curve occurs when test cases are run that were not anticipated by the programmers. However, their analysis is not linked specifically to the test techniques discussed in this report.

One way of estimating the number of errors in a program is by using the technique of error seeding. To estimate the number of errors in a program, one can artificially introduce new errors (by changing correct program statements) and then apply an error detection process.

[1]M. R. Paige and E. E. Balkovich, A Test Plan for a Structured Program, General Research Corporation RM-1638/1, May 1972, p. 25.

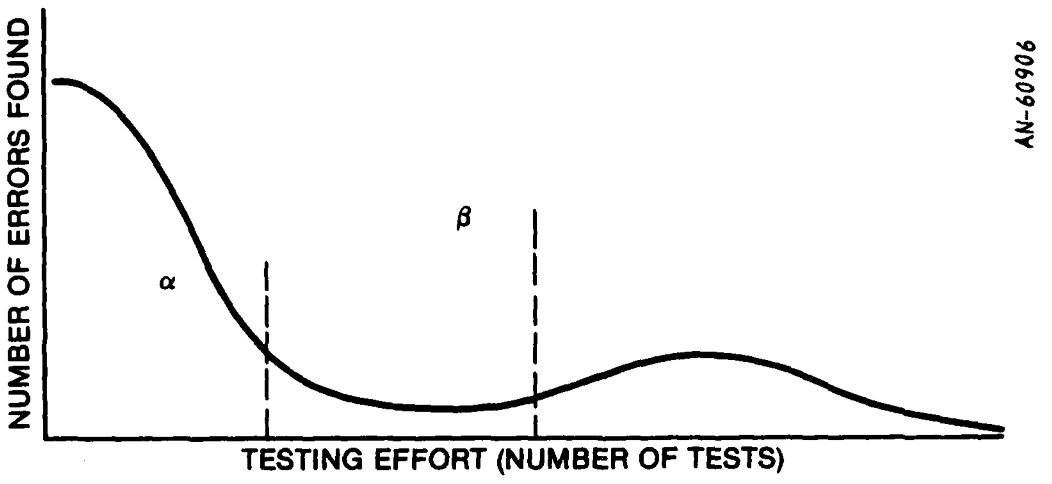

Figure 3.1. Marginal Test Benefit as Hypothesized by Paige and Balkovich

A maximum-likelihood estimate of the number of errors in the program is then:[1]

$$\left[\frac{n \times (r - k)}{k}\right]$$

where	r	is the number of statements in the program,
	n	is the number of errors introduced,
	k	is the number of errors detected (real and seeded),
	[ ]	indicates the greatest integer function.

[1]G. J. Schick and R. W. Wolverton, "An Analysis of Competing Software Reliability Models," IEEE Transactions on Software Engineering, Vol. SE-4, No. 2 (March 1978), pp. 112-114.

Another way to estimate the number of errors in a program is to use a complexity metric. No firm analytical basis exists for the use of a particular complexity measure, since no single aspect of program structure can explain the behavior of the entire program. Nevertheless, a considerable body of data supports the use of complexity measures as error estimators.

Halstead[1] developed a set of metrics which are based on counts of the number and incidence of operators and operands in a program. He proposes an "effort" metric E, which is to be proportional to the number of errors in a program. Effort is estimated from the operator and operand count by the formula:

$$E = \frac{(N_1 + N_2) \times \log (n_1 + n_2)}{(2/n_1) \times (n_2/N_2)}$$

where	N_1	=	total number of occurrences of operators
	N_2	=	total number of occurrences of operands
	n_1	=	number of distinct operators
	n_2	=	number of distinct operands

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

Lloyd and Lipow[1] describe a model which estimates the mean time to next failure of a program. The model assumes a Poisson distribution for the number of errors detected during a time interval of the testing process, with the mean of the distribution being proportional to the number of errors in the program at the beginning of that interval. The mean time to failure for the program is computed as:

$$\theta = \frac{1}{W \times (N - n)}$$

where θ = mean time to failure

W = the proportionality constant for the number of errors detected (calibrated for the program)

N = estimated number of errors in the program

n = number of errors detected in the test interval

3.3 MEASURING TESTING COSTS

We look at two areas in our analysis of the costs of software testing: the direct co^ts of applying the individual techniques, and the cost savings achievable by using the techniques. The direct costs of automated testing include the human skills and time, computer resources, and tool procurement or development costs. Potential cost

[1] D. K. Lloyd and M. Lipow, Reliability: Management, Methods, and Mathematics, Second Ed., 1977, pp. 514-521. Published by the authors, who are with TRW Systems and Energy Division, Redondo Beach, California.

savings are analyzed in terms of the early detection of errors and increased testing efficiency that automated techniques can provide.

We cannot give estimates of how much, in dollars, it costs to use the techniques. This is because dollar costs depend upon the tool being used and the particulars of the test environment. However, we do provide "cost worksheets" as part of the characteristic profiles of each technique in Chapter 5. These worksheets give the prospective test technique user a way to form an estimate of testing costs, based on his knowledge of the size of his testing problem and the per-unit costs of human and computer resources.

3.3.1 User Skills and Time

The skills required to use a test technique indicate the professional level that the tester must have. For example, most static testing tasks can be performed by someone with a rudimentary programming background and no knowledge of the application area of the program being tested. On the other hand, formal verification requires a high degree of familiarity with proof techniques and a detailed knowledge of the program being tested and its application area.

We have accumulated a few pieces of data on the amount of analysis time used during applications of the test techniques. There is not enough data to develop with confidence any "average analysis time" values. We have included the available data in the characteristic profiles in Chapter 5 to reflect some people's experiences with the test techniques.

3.3.2 Computer Resources

The computer resource requirements of automated test techniques take two different forms. In the cases of static analysis and formal testing, the source code of the program being tested is operated upon by the test software. Thus, the overhead of these techniques is the cost of running the test tool.

On the other hand, dynamic testing requires repeated executions of a version of the program being tested that may have been instrumented or had assertions placed in it. The number of runs made in dynamic testing depends on how thorough the user wants the testing to be and on how many errors are in the program.

Some standard is needed to make comparisons of computer costs across different computer systems and target languages. It seems natural to compare the cost of static analysis to that required to compile the same program. Similarly, the cost of making a dynamic test run can be compared to running an unaltered version of the same program. The compilation and execution time benchmarks yield good approximations, but not exact formulas, for the computer resources used in testing.

Size statistics on automated test tools include the amount of memory required to execute the tool, the number and size of temporary and permanent data bases, and the tool's input and output characteristics. Although this information is highly tool-dependent, there are similarities in the operation of the tools implementing each technique. We present the size statistics of a few representative tools as part of each characteristic profile.

3.3.3 Tool Procurement Costs

The cost of acquiring a test tool may be a small part of total testing outlays if a user expects the tool to be employed in a large number of test efforts. However, tool procurement may require a large initial expense under certain conditions. Test tools have not been built for all possible combinations of test techniques, computer systems, and target languages. If no tool exists for a computer and language similar to what a user desires, he must assume that his tool procurement costs will be nontrivial.

3.3.4 Cost Savings Provided by Automated Techniques

Lipow[1] presents a simple cost tradeoff model of the decision as to whether to use a test tool. According to the model, the use of the tool is worthwhile if:

$$C_A < P \times N \times (C_O - C_T)$$

where C_A = cost of acquiring and applying the tool

P = additional proportion of errors discovered by the tool

N = number of errors detected by testing without the tool

C_O = average cost of detecting an error during the operations phase

C_T = cost of detecting an error during testing

[1] M. Lipow, "Prediction of Software Failures", Journal of Systems and Software, Vol. 1, No. 1 (1979), pp. 74-75.

4 COMPARISONS OF THE SOFTWARE TESTING TECHNIQUES

Good software testing techniques should be effective, reliable, inexpensive, and easy to use. This report explores how good current techniques really are. In this chapter we make some general comparisons of the four test techniques used for error detection, along with formal verification. The comparisons are based on the metrics discussed in the last chapter and on some other criteria presented here. In Chapter 5 we look at each technique in depth.

We have ranked the five test techniques from best to worst for many of the criteria. We do not designate a most effective test technique; effectiveness is a complicated and controversial subject. Static analysis is rated the most reliable, least expensive, and easiest to use of the techniques.

We must point out that static analysis is the most fully-developed of the test techniques. A fairly large number of static analysis packages have been in use since the mid-1970s. Meanwhile, methodologies for structural, functional, and executable-assertion testing are still being developed, and fully automated implementations of these methods have not yet appeared. Formal techniques have survived a period in which their legitimacy was under attack, and practitioners are just beginning to develop formal tools for use outside the laboratory.

4.1 EFFECTIVENESS

We use two indicators of the effectiveness of a test technique:

- The range of error types that it can detect
- The percentage of all errors in a program that it actually does detect

In Table 4.1 we display our rank-ordering of the static and dynamic test techniques, based on the range of error types that they can detect. Formal verification is omitted because we are not sure how a proof system would cope with all of the types of errors in the TRW scheme. Our treatment of effectiveness in evaluating formal verification is different from that used for the other techniques. The major TRW error categories addressed by each test technique are listed in Table 4.2.

TABLE 4.1

RANKING OF THE TECHNIQUES BASED ON RANGE OF ERROR TYPES DETECTED

(1 = best)

1. Functional testing
2. Executable assertions
3. Structural testing
4. Static analysis

We have used a great deal of judgment in evaluating the range of errors detected, especially for the dynamic test techniques. Most program errors can be detected by dynamic testing if a test case that reveals the errors happens to be selected. We tried to determine whether each type of error in the TRW scheme fits the "purpose" of the test technique being evaluated. This is especially hard for functional testing, since functional testing can be roughly described as finding input data that is likely to cause the program trouble. We rate functional testing very highly on the range of errors that it can detect--but we also feel that some experiments need to be done to verify this.

TABLE 4.2

MAJOR TRW ERROR CATEGORIES ADDRESSED BY TEST TECHNIQUES

TRW Major Error Categories	Static Analysis	Executable Assertions	Structural Testing	Functional Testing
Computation Errors		x	x	x
Logic Errors		x	x	x
Data Input Errors		x		x
Data Handling Errors	x	x	x	x
Data Output Errors			x	x
Interface Errors	x	x		x
Data Definition Errors	x	x		x
Data Base Errors	x	x		x
Operation Errors				x
Documentation Errors	x			x

In Sec. 3.2.2 we discussed the use of an error detection ratio metric. This metric, our second indicator of test technique effectiveness, is the ratio:

$$\frac{\text{Number of errors detected}}{\text{Total number of errors in a program}}$$

At present, no one knows enough to predict that a test technique will detect a particular minimum, maximum, or average percentage of the errors in a program. While we believe that, generally, static analysis will detect fewer errors than dynamic testing, there may well be cases where this is not so. We do not rank-order the techniques on the basis of percentage of errors detected, because current knowledge is insufficient to support such a conclusion.

4.2 RELIABILITY

We gauge the reliability of a technique by the variation in its effectiveness over different testing applications. There are three main causes of such variation: the human factors involved in testing, the kinds of errors that the techniques detect, and the characteristics of the programs being tested. We discuss these causes in Chapter 5 but we also want to highlight them here.

Of the five techniques, static analysis is least susceptible to human error, is most consistent in applying error checks, and varies least in the way it works for different kinds of programs. We have declared it to be the most reliable technique on this basis. However, the reliability of all the techniques, including static analysis, it less than ideal due to the following factors:

- All of the techniques are susceptible to human error and abuse.

- There are very few error types (from the TRW classification scheme) that can be eliminated with certainty by any type of testing or program proving.

- Data on the use of the test techniques shows a large variation in the percentage of errors that are detected in different programs.

4.2.1 Human Factors Involved in Testing

Three things are required of a person who is using any of the test techniques:

- He must have some knowledge and control of the program development process in order to plan and conduct tests.
- He must carry out the mechanics of using the tools.
- He must use skill and judgment to help in the error detection process.

Any of these areas may be sources of problems which reduce the effectiveness of the techniques. These problems can be reduced by making test techniques flexible, easy to use, and highly automated, and by developing standard methodologies to support them.

In Table 4.3 we rank the techniques according to their sensitivity to human factors. Static analysis is much less sensitive to human errors and limitations than the other techniques. Static testing is the most automated technique and has the most developed methodology. Most of the elements of skill and judgment have been removed from static testing because the error checking is easily understood and automatically applied.

TABLE 4.3

SENSITIVITY OF THE TECHNIQUES TO HUMAN FACTORS

(1 = Least sensitive)

1.	Static analysis
2.	Structural testing
3.	Functional testing
4.	Executable assertions
5.	Formal verification

Testers have varying degrees of influence over project management, and this can affect the reliability of the tests. For example, a programmer who needs to check out his own code may not have access to other modules. This makes static global data flow analysis *impossible*; it can also make functional testing difficult or meaningless. In general, the following things need to be under control of the tester for effective use of the test techniques:

- Static analysis needs information about the behavior of global variables. This may come from design documents, module stub libraries, or code.

- Dynamic testing requires that the tester be able to determine whether the code is operating properly. The test environment needs to be as similar to the operating environment as possible -- realistic input data should be used, all interfaces should be simulated, and other special conditions (for example, timing considerations) need to be provided.

- Formal verification involves proving the consistency of the code and its specifications. Changes will need to be made in both code and specifications to allow a proof to be completed. Formal verification needs to be planned intensively from the beginning of a project.

In Sec. 4.4 we discuss the degree of automation and the user skill requirements of the test techniques. The more automated a technique is, the less opportunity there is for human mistakes to corrupt the results of a test. Automation also increases the thoroughness of testing and ensures that proper methods are adhered to.

4.2.2 Kinds of Errors Detected

In the characteristic profiles we discuss the error types for which the techniques are most and least effective. We also examine the reliability of each technique for those error types that it detects most effectively.

If a technique is completely reliable for an error type, then testing with the technique can guarantee that no errors of that type are present in a program. However, very few error detection methods can find errors with 100% certainty. This is true even of static analysis, which is usually thought of as highly reliable. For example, static data flow analysis produces very weak results for subscripted variables (see Sec. 5.1.3).

If given enough information, static testing can detect all occurrences of the following error types:

- Module interface errors
- Coding standards violations
- Mixed-mode arithmetic
- Unreferenced statement labels

We know of no other error types that can be detected with complete certainty by any type of testing. It would be nice to associate a probability measure with error detection. Such a metric would complement the error detection ratio as a gauge of technique effectiveness. However, not enough is currently known about error detection to do this.

4.2.3 Program Characteristics

In the characteristic profiles we look at how the test techniques are affected by some characteristics of the program being tested. These characteristics include:

- The life cycle phase in which testing is being conducted
- Program size and complexity
- The type of program and its intended application (numerical or nonnumerical, real-time or noncritical, etc.)
- The language and design methods used

In Table 4.4 we indicate the life cycle phases in which the use of each test technique is appropriate. The fact that one technique must be used later in the life cycle than another does not imply that it is inferior. Testing is needed throughout the development period, and different techniques should be called on at different times. But since it is cheaper to correct errors earlier in the life cycle, testing should begin as early as possible.

The other program characteristics cannot be treated as neatly as life cycle phase. The assertion testing experiment (Appendix D) investigates the relationship between program complexity and testing. In the technique profiles we comment on the usefulness of the techniques for certain application areas. We also mention a few ways that advanced programming languages and design techniques have facilitated testing.

TABLE 4.4

APPLICATION OF TEST TECHNIQUES IN THE LIFE CYCLE

	Coding and Checkout	Test and Integration	Installation	Operations and Support
Static Analysis	X	X	X	X
Executable Assertions	X	X		X
Structural Testing	X	X		
Functional Testing		X	X	X
Formal Verification	X	X		

Researchers have only recently begun to look at the problems of testing real-time and distributed systems. Such programs have special error conditions besides those that are found in non-time-critical programs running in a single-processor environment. The five test techniques described in this report do not address the problems of synchronizing concurrent processes, allocating shared resources, and many other important issues. However, some progress has been made in applying the test techniques to other than conventional programs:

- Taylor and Osterweil[1] describe how the techniques of static data flow analysis can be extended to detect certain error and anomaly types in concurrent programs.

- Executable assertions can provide fault-tolerance for programs susceptible to hardware and communication (as well as processing) errors. Assertions can also provide security during sharing of resources.

- Instrumentation tools, which give coverage data to support structural testing, can be used to measure the speed and frequency of execution of program segments. This information can be used to improve the efficiency of time-critical code.

- Formal verification has been used to establish the security of the communications in a computer network (see Sec. 5.5). Proof of other properties of distributed and time-critical systems is an active research area.

4.3 COST

This report has two goals in evaluating the cost of test techniques. First, we want to develop a fairly complete list of the resources required to use a technique. Second, we present some data from actual test experiences to give a general idea of the significance of the resources.

[1] R.N. Taylor and L. J. Osterweil, "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," IEEE Transactions on Software Engineering, Vol. SE-6, No. 3 (May 1980).

It is generally agreed that static analysis is the cheapest test technique to use, with the dynamic techniques being significantly more expensive, and the formal techniques much more expensive than the dynamic techniques. In Figure 4.1 we show an order-of-magnitude comparison of the costs of static, dynamic, and formal testing using current technology.

One reason for the cost differences is the nature of the techniques: static testing requires only a single execution of a source code analyzer; while dynamic testing requires repeated executions of the test program and analysis of each set of test results; and formal techniques involve difficult intellectual labor with limited mechanical support. However, static analysis is also the most fully-developed of the test techniques. We predict that in the next ten years static testing will remain the cheapest technique, but the cost gap between it and the other techniques will close to some extent.

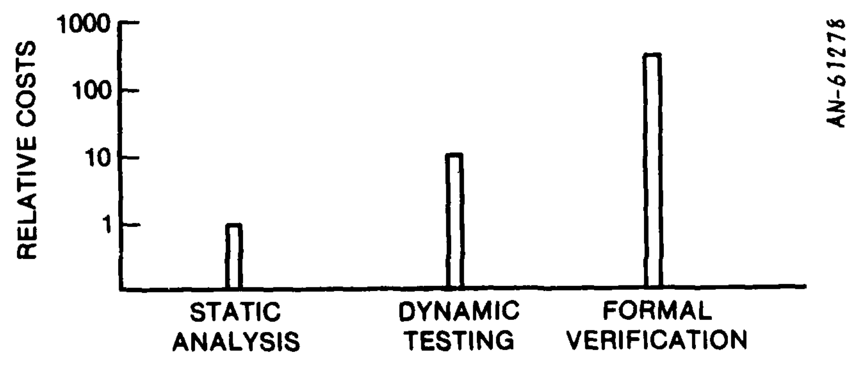

Figure 4.1. Relative Costs of Static, Dynamic, and Formal Testing

4.3.1 User Skills and Time

Although modern software testing tools are highly automated, some techniques still require special skills and a significant degree of user involvement for their application. The most completely automated techniques are those of static analysis. Static error detection, anomaly detection, static assertion checking, and code auditing require little familiarity with the program under test or its application area. Static analysis can be easily applied, for example, by an independent testing team without thoroughly understanding the program or the problem it solves.

Structural testing requires strong programming skills to analyze test results and devise new test data. A deeper understanding of the application problem area also helps but is not essential. Path testing, for example, identifies the program paths traversed by a set of test input data. The tester must be able to determine how to adjust the input data to exercise and test the remaining paths.

Functional testing, executable assertions, and formal program verification require a much more complete understanding of the application problem area and the program's requirements and specifications. This is in addition to the programming skills described above. Functional testing requires analysis of the domains of all input data and identification of special values and special combinations of functionally related values. Identifying loop-invariant relationships for executable assertion testing is equally difficult. Formal proof systems are very complex and require extensive training to be used effectively.

4.3.2 Computer Resources

The automated techniques require a greater amount of computer resources than the manual test techniques that have been traditionally used. The extra computer costs must be weighed against the benefits of more thorough testing and the manpower savings that automated techniques can provide. The computer resource overhead required by a technique can be tool-dependent and problem-dependent.

Static analysis requires the application of a code analyzer which may also build a data base for use by other automated tools. One application of the static analysis tool is necessary per code version tested. The computer time and storage needed by static tools used to be a serious impediment to their use on large programs. However, now there are tools available whose resource requirements are linear functions of program size.

Executable assertion testing, instrumentation, and test harnesses all require that additional code be inserted into a program before it is executed. The overhead required by assertions and test harnesses can vary a great deal. The overhead required by instrumentation tools depends on the level at which instrumentation is performed (statement, branch, or module) and the size and complexity of the code. The data cited in Secs. 5.2 suggest that assertions require from 0% to 50% increase in program execution time and about a 10% to 15% increase in storage. Instrumentation tools generally require a 2% to 50% increase in execution time and a 20 to 100% increase in program size.

Functional testing may be conducted without automated assistance, so the minimum execution time and storage overhead for this technique is zero. However, functional testing requires a large number of program executions, so computer time costs and restrictions are important factors in its use. The number of executions required to perform functional testing depends on the degree of testing completeness desired, the nature of the errors in the program, and the difficulty of detecting and correcting errors.

The amount of computer resources used by the formal techniques is highly dependent upon the difficulty of obtaining the results desired.

Some tools will continue indefinitely to seek a needed verification condition or expression simplification, while others will stop and await new information from the user. The tools which support the formal techniques are typically written in LISP and use garbage collection algorithms during execution — thus they can be very slow. The application of formal techniques to a medium-sized or large program is likely to require a significant amount of computer, as well as human, time and resources.

4.3.3 Procurement Costs

The availability of automated testing tools for specific programming languages and host computer systems is growing. As Table 4.5 shows, most currently available tools are for ANSI FORTRAN. Static analysis and instrumentation are the most common techniques supported by the tools surveyed. Table 4.5 reflects the predominance of tools developed for CDC and IBM host computers as well as the predominance of FORTRAN. New testing tools are being developed to fill in obvious gaps in the current tool availability picture. However, the balance of test tool availability is expected to remain high for FORTRAN and for the major computer manufacturers.

4.3.4 Cost Savings Provided by Automated Techniques

Several studies have indicated that the use of automated test tools can result in substantial cost savings over traditional manual testing methods.

- Alberts[1] reports that "the use of automated instruction and path checkers ... catch between 67% and 100% of the errors (in a program) and between two to five months earlier than they would otherwise have been detected." He estimates the

[1]D. S. Alberts, "The Economics of Software Quality Assurance," National Computer Conference, New York, AFIPS Press, June 1976, p. 441.

TABLE 4.5

TEST TOOL AVAILABILITY BY TEST TECHNIQUE AND PROGRAMMING LANGUAGE

		ANSI FORTRAN	Structured FORTRAN	JOVIAL	COBOL	All Others
1.	Standards	3	0	1	0	0
2.	Static Analysis	12	3	2	1	3
3.	Test Data Generation	2	1	0	0	2
4.	Test Harness	4	0	0	3	2
5.	Instrumentation	7	2	1	2	2
6.	Debugging Aids	1	0	0	3	1
7.	Dynamic Analysis	5	2	3	0	0
8.	Symbolic Execution	5	1	1	0	4
9.	Formal Verification	0	0	0	0	6
10.	Mutation Analysis	1	0	0	0	0

TABLE 4.6

TEST TOOL AVAILABILITY BY HOST COMPUTER MANUFACTURER AND PROGRAMMING LANGUAGE

Host Computer Manufacturer	FORTRAN	COBOL	JOVIAL	All Others
CDC	13	0	3	1
Honeywell	4	1	1	0
IBM	14	3	1	4
Univac	7	1	0	0
All Others	8	1	1	4

total cost savings by the use of automated tools in a half-billion dollar software development project as $25 million, due to productivity increases averaging 10%. (This assumes that 50% of the project costs occur during the development phase.)

- Deutsch[1] has claimed an overall saving in testing costs from using structural testing. The money saved is due to a higher error detection rate early in the development cycle by using a tool, as shown in Fig. 4.1. He estimates a net saving of over five times the cost of using the tool.

- The increase in test efficiency and effectiveness of a specific automated tool is documented by Brown, et al.[2] The instrumentation tool PACE was used to analyze a large flight trajectory program that had previously been tested without automated aids. The earlier testing process had produced 33 test cases which exercised only 85% of the subprograms in the package. PACE helped to identify a set of six test cases which exercised 93% of the subprograms. Use of the smaller set of test cases provided for increased coverage while significantly reducing the time required to perform the tests.

[1]M. S. Deutsch, "Software Project Verification and Validation", Computer, April 1981, pp. 54-70.

[2]J. R. Brown, A. J. DeSalvio, D. E. Heine, and J. G. Purdy, "Automated Software Quality Assurance", in Program Test Methods (W. C. Hetzel, ed.), Prentice-Hall, 1973, pp. 201-202.

4.4 EASE OF USE

A summary of the difficulty of using the test techniques is presented in Table 4.7. The table rates the techniques in the categories of user skills required, degree of user involvement, and analysis required to detect and locate errors. The skills required to use the test techniques were described in Sec. 4.3.1 above and in Chapter 5. The way the techniques help to locate errors is discussed in Sec. 6.1.

Static analysis is the easiest technique to use and is rated low in all three categories. The most time-consuming chore associated with static testing is sorting out the extraneous warning messages that are generated for data flow and mixed mode anomalies. Extraneous warnings are a significant problem in static testing, but the analysis required for other forms of testing is still much greater.

The dynamic test techniques are rated more difficult than static analysis in all three categories. To perform dynamic testing, the user has to manually prepare sets of input data and, in the case of assertion testing, add statements to the source code of the test program. To formulate effective assertions that check computations or to develop sets of input data that test boundaries of program function domains, a tester needs a high degree of understanding of a program's requirements and principles of operation. In the case of structural testing, the path analysis and coverage information that tools provide can substitute for such expertise.

Neither functional nor structural testing have any automatic mechanism for indicating that an error has occurred during a test run. Assertion violations are called to the tester's attention by a printed message, but the cause of the violation is not necessarily close to the location of the violated assertion.

TABLE 4.7

EASE OF USE RATINGS

TECHNIQUE	USER SKILLS	DEGREE OF USER INVOLVEMENT	ANALYSIS FOR ERROR DETECTION AND LOCATION
STATIC ANALYSIS	L	L	L
EXECUTABLE ASSERTIONS	H	M	M
STRUCTURAL TESTING	M	M	H
FUNCTIONAL TESTING	H	M	H
FORMAL VERIFICATION	VH	H	H

EXPLANATION OF RATING SYSTEM

User Skills

L	- Familiarity with the program under test or its application area is not required. Testing could be conducted by a party independent of the programming team with little documentation or training.
M	- Programming skills and familiarity with the structure of the test .am are required. Expertise in the program's application area not required.
H	- Specialized programming and testing skills are required. Familiarity with the program's application area is very helpful for effective testing.
VH	- Uses specialized mathematical techniques that may be unfamiliar to a programmer.

Degree of User Involvement

L	- The testing process (test initiation and execution) is fully automated. No manual preparation of inputs or source code or guidance of test execution is necessary.
M	- Some manual preparation is required before a test can be performed. The test itself is done automatically.
H	- User must manually operate the test tool during performance of the test.

Analysis for Error Detection and Location

L	- Most of the error types detected are indicated automatically. In addition, the source or location of a large number of errors is identified.
M	- Errors are detected automatically. User analysis is usually required to locate the cause of the error.
H	- User analysis is required to both detect errors and locate them.

Formal testing requires a great amount of user effort and analysis. Operating an algebraic expression simplifier or developing formal verification conditions are specialized skills not commonly possessed by those who write and test applications programs. Formal techniques are not designed to simplify the process of error detection. Their purpose is to make it possible to determine rigorously that a program satisfies its specifications.

PRECEDING PAGE BLANK-NOT FILMED

5 PROFILES OF THE TECHNIQUES

5.1 STATIC ANALYSIS

5.1.1 Summary of the Technique

Static analysis is a collection of analysis and testing methods that do not require the execution of the subject program. Static analysis can identify errors, enforce good coding practices, and provide information that is useful for dynamic testing and program maintenance. Static analysis is almost completely automated, so it is easy to use. There are static tools available for most programming languages and computer systems.

Capabilities

Static analysis includes any method of testing which involves only the examination of program source code. Static analysis can accomplish several things:

- It can detect and locate certain types of program errors.
- It can identify program anomalies—characteristics that in some cases produce errors.
- It can identify constructions that do not conform to a standard syntax.
- It can determine whether variables are used in accordance with the programmer's intentions.
- It can help to generate test data for dynamic tests.
- It can provide documentation reports.

In Sec. 2.1.1 we presented five types of programming errors that can be detected by static techniques. The errors are detected by analyzing the control structure and data flow of the source code. These errors are:

- Structurally infinite loops--loops which provide no possibility of termination because there are no exit points
- Module interface conflicts--mismatch of actual and formal parameters
- Recursive procedure calls, either direct or indirect
- Uninitialized variables
- Structural deadlock in concurrent programs

Five types of programming anomalies can be detected by tools which examine statement syntax, analyze control structure and data flow, and tabulate program statistics. These activities are necessary for other static analysis functions (error detection and documentation), so anomaly detection does not require a major extension of the capabilities of a tool. Anomaly detection increases the number of errors that static testing can find; however, the user must determine which identified anomalies are in fact errors. The five anomaly types are (see Sec. 2.1.2):

- "Questionable" coding practices, such as over-use of GOTOs
- Mixed-mode expressions
- Data-flow anomalies: variables set and not used, extraneous variables, consecutive assignments to a variable without intervening use
- Structurally unreachable code
- Unreferenced statement labels

Static analysis detects instances, rather than symptoms, of most errors and anomalies. It also gives their location, which makes static tools very useful for debugging as well as testing programs. We discuss the debugging capabilities of static analysis in Chapter 6.

Compilers enforce the syntax standards of a language by rejecting code that has unacceptable constructs. However, compilers usually represent "dialects" of a language, which may include features that are peculiar to one host machine or operating system. Special tools have been developed to ensure that a program complies with a more rigorous set of language standards, so that the program can be used in a wider range of environments. A few examples of such "standards enforcement" tools are found in the tools survey in the first interim report.

Two types of assertions can be checked by static techniques: variable usage assertions and units assertions. Variable usage assertions describe how a variable is intended to be used in a module: strictly as input, strictly as output, or both as input and output. The static tool can check to see if the actual use of the variable is the same as what was intended. Units assertions tie units (feet, volts, dollars, etc.) to variables. A tool can check to see that expressions that must agree in unit (e.g. both sides of assignment statements) do agree after algebraic simplifications are made.

Two kinds of test data generation capabilities can be built into a static tool. One kind is mainly used in COBOL-based tools: the tool identifies the names and types of input variables, and determines input file format information from program input statements. Actual values for the input variables are then selected randomly.

The other static test data generation method is an adaptation of symbolic execution. Input values which will cause a given program path to be executed can be obtained by examining the predicate conditions along the path. Automated methods of determining the predicate conditions can produce satisfactory results in some cases, but not always.

We have identified six types of documentation reports that static tools generate. Such reports are useful for building "off-line" documentation (reference manuals, user guides) for programs. They also provide invaluable aids to maintenance and modification, especially for large programs. The six report types include:

- Variable cross-reference reports
- Module invocation reports
- Module interconnection matrices
- Global data reports
- Program statistics
- Summaries of all static analysis functions performed

<u>Operation</u>

Static analysis is a very easy test technique to use. All of the static functions described above, except for assertion checking, can be performed on a program without any preparation or modification of the source code whatsoever. The only inputs required by a typical static tool are the test program's source code and a small set of instructions controlling the operation of the tool. Testing is fully automated, except for the analysis required to determine errors from detected anomalies.

Static analysis is also very flexible, since it can be used on amounts of code ranging in size from one module to a very large program. Parts of programs can be tested and a full analysis of global properties still performed; this is done by using "stub module libraries",[1] which supply information about missing modules.

[1] The use of stub libraries in the SQLAB tool is described in S. H. [illegible] et al., <u>Advanced Software Quality Assurance Final Report</u>, General Research Corporation CR-3-770, May 1978, pp. 143-144.

Automated Tool Support

There are a large number of static test tools available. These tools cover a fairly wide range of languages and computer systems. Table 5.1 summarizes the static tools that were surveyed in the first interim report.

5.1.2 Effectiveness

Static testing can detect a surprisingly wide range of error types. It is best at catching data handling and interface errors, worst at finding computational, logic, and input/output errors. Data on the percentage of program errors detected by static analysis ranges from 16% to 55%.

Types of Errors Detected

The following error types from the TRW error classification system have counterparts in the static error and anomaly checks listed above.

- A_400 Units or data conversion errors
- A_700 Precision loss due to mixed mode
- D_100 Data initialization not done
- D_600 Incorrect variable type
- F_300 Subroutine arguments not consistent in type, units, order, etc.
- F_400 Subroutine called is nonexistent

There are other TRW error categories that may be caught by static testing in some cases. Many of these categories describe errors that may be associated with data flow anomalies or structural control flow problems (infinite loops, dead code). However, these error types cannot always be detected statically. For example, the "A_100 Incorrect operand in equation" error can be caught by data flow analysis if the operand used is uninitialized, but probably won't be otherwise. These error categories include:

TABLE 5.1
STATIC TEST TOOLS
SOURCE: Tools Survey, Appendix B

TOOL	LANGUAGES	ERROR AND ANOMALY DETECTION	STANDARDS ENFORCEMENT	ASSERTIONS	TEST DATA GENERATION	DOCUMENTATION
ACES	FORTRAN	✓	✓		✓	✓
AMPIC	FORTRAN	✓				✓
ATDG	FORTRAN	✓			✓	✓
AUDIT	FORTRAN	✓	✓			
CAVS	COBOL	✓		✓	✓	✓
DAVE	FORTRAN	✓				
FACES	FORTRAN	✓	✓			✓
FAVS	FORTRAN	✓			✓	✓
J73AVS	JOVIAL J73	✓	✓	✓	✓	✓
PDS	EL1	✓				
PFORT	FORTRAN	✓	✓			✓
RXVP80™	FORTRAN	✓				✓
SADAT	FORTRAN	✓			✓	✓
SMOTL	SMOD	✓			✓	
SQLAB	FORTRAN, PASCAL, JOVIAL	✓		✓	✓	✓
SURVAYOR	FORTRAN	✓				✓

- A_100 Incorrect operand in equation
- A_800 Missing computation
- B_100 Incorrect operand in logical expression
- B_200 Logic activities out of sequence
- B_300 Wrong variable being checked
- B_400 Missing logic or condition tests
- B_600 Loop iterated incorrect number of times (including endless loop)
- D_200 Data initialization done improperly
- D_300 Variable used as a flag or index not set properly
- D_400 Variable referred to by the wrong name
- F_100 Wrong subroutine called
- F_200 Call to subroutine not made or made in wrong place
- G_100 Data not properly defined or dimensioned
- H_300 Data units are incorrect
- J_500 Code or design inefficient or not necessary

We found two studies which discuss the effectiveness of static test techniques for different error types. The TRW report presents an analysis of how many errors in the Project 3 study would have been caught by four types of static test tools.[1] The tools that they considered are:

- Code standards auditor
- Units consistency analyzer (processes units assertions)

[1]Pages 4-168 through 4-171 of the TRW Report. Project 3 used an error classification scheme that was later changed slightly to form the Project 5 scheme which we discussed in Sec. 3.2.1.

- Set/use checker (global data flow analysis)
- Compatibility checker (examines calling sequences for errors)

TRW determined that these four types of tools could detect significant percentages of errors[1] in the following categories for one software project:

- Computational (15%)
- Logic (9%)
- Input/Output (17%)
- Data handling (44%)
- Routine/Routine interface (78%)
- Routine/System software interface (72%)
- Data base interface (100%)
- Global variable/compool definition (62%)

The ability of the SQLAB tool to detect different types of errors was also analyzed by GRC.[2] The error types considered are from an error classification scheme developed by Logicon.[3] The GRC report describes ways that static testing (and other test techniques) might cope with each error type. The findings are summarized in Table 5.2.

<u>Error Detection Ratio Data</u>

We found six sources of data on the percentage of errors in a program that were detected by static testing.

[1]The error categories listed are those for which at least one tool would catch 9% or more of the total errors in the category. See Table 4-34 on p. 4-169 of the TRW Report. This study is an example of historical data gathering so the actual percentages are somewhat speculative.

[2]S. H. Saib, et al., Advanced Software Quality Assurance Final Report, General Research Corporation CR-3-770, May 1978, pp.111-122.

[3]J. A. Dana and J. D. Blizzard, Verification and Validation for Terminal Defense Program Software, The Development of a Software Error Theory to Classify and Detect Software Errors, Logicon HR-74012, May 31, 1974.

TABLE 5.2

STATIC ERROR DETECTION METHODS (SQLAB TOOL)

	Error Type	Detection Method
1.	Data/instruction access and storing	Data flow analysis, mode/type checking, and units assertions can catch wrong variable names. Data flow analysis and documentation review can catch missing COMMON statements.
2.	Equation computation and arithmetic	Units assertions can catch some incorrect operators.
3.	Branch and jump	Misplaced statement label may result in structurally infinite loop or dead code.
4.	Incorrect constant value and data formats	Type checking identifies inconsistent data types. Data flow analysis detects undefined and multiply-defined data.
5.	Specification violation due to incorrect implementation	Documentation review and module interface conflict checking can detect missing or extra modules and incorrect use of routines.
6.	Incomplete or erroneous specifications	Units assertions can catch dimension errors in equations.
7.	Logic and sequencing	Documentation and data flow analysis can sometimes catch out-of-sequence operations.

- Rubey, et al.[1] studied several software projects. Of a total of 378 errors found during the development and validation phases of the projects, static analysis caught 167, or 44%.

- GRC[2] conducted an experiment using a FORTRAN missile trajectory program with 57 modules and 5000 lines of code. Forty-nine errors were seeded into the program and the SQLAB tool's static test capabilities were used. Eight errors (16%) were detected.

- As a preliminary to that study, GRC looked at eight small (less than 30 source lines) programs from Kernighan and Plauger.[3] These programs contain 26 "typical" programming errors. Static testing with SQLAB found 10 errors (38%), and testing with DAVE (data flow analysis only) found 8 errors (30%).

- TRW estimated that a code standards auditor would have caught 26.3% of all errors in the Project 3 study.[4] They also estimated that a set/use checker would catch 14.3% and a compatibility checker 10.7% of the errors in Project 3. They made no estimate of the combined effectiveness of these tools.

[1] R. J. Rubey, J. A. Dana, P. W. Biche, "Quantitative Aspects of Software Validation", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2 (June 1975), p. 153.

[2] C. Gannon, R. N. Meeson, N. B. Brooks, An Experimental Evaluation of Software Testing, General Research Corporation CR-1-854, May 1979, p. 5-8.

[3] B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, First Edition, McGraw-Hill, 1974. The results of the GRC experiment are cited on pp. 2-1 and 2-2 of Gannon, et al.

[4] Pages 4-169 and 4-170 of the TRW Report contain this data.

- Howden[1] studied the errors found in a release of the IMSL Scientific Subroutine Package. He determined that 27 of the 49 errors, or 55%, could have been caught by static analysis techniques.

- Howden[2] also conducted a study of six programs written in four different programming languages. He states that two of the 28 errors present could be caught by checking routine interfaces, and four could be caught through anomaly analysis. This represents 21% of the total errors in the programs.

5.1.3 Reliability

Static analysis is highly automated, so error checking is applied consistently. Because it is automated, human error is unlikely to greatly corrupt the testing process. But static testing has inherent limitations; there are few errors from the TRW classification that it can catch with a high degree of consistency. It is usually most effective during the early stages of program development. Static testing can be used with equal ease on large and small programs. The features and standards of new programming languages make some static checks unnecessary.

Human Factors

Since static analysis is highly automated and easy to use, the abilities of the tester are not as important to the reliability of the technique as is the case for dynamic and formal testing. The mechanics of using static tools are simple; tests can be conducted by someone

[1]W. E. Howden, Effectiveness of Program Validation Methods for Scientific Programs, National Bureau of Standards GCR 78-148, 1978, p. 53.

[2]W. E. Howden, "Theoretical and Empirical Studies of Program Testing", IEEE Transactions on Software Engineering, Vol. SE-4, No. 4 (July 1978), p. 296.

unfamiliar with the code being tested. All error and anomaly checking is performed by the computer, so the degree of thoroughness and attention to detail is much greater than what could be achieved manually.

There are two areas in which human factors can influence the success of static testing. First, as mentioned in Sec. 4.2.1, global data flow analysis needs information about the behavior of variables in all modules of a program. This information may be unavailable under certain circumstances; for example, to a programmer who wants to check out new code for a small part of a large project. However, the information could be made available from sufficiently detailed design documents.

Skill and judgment is involved in sorting out errors from the warning messages produced by anomaly checking. Sometimes it is easy to decide whether a message indicates an error is present; but sorting through a long list of such messages can be tedious, and there is the danger that an important warning will be overlooked when a lot of extraneous ones are discarded. Data-flow and mixed-mode anomalies can be symptoms of subtle errors, and should get more than cursory attention from the tester.

<u>Kinds of Errors</u>

Because they are automated, static tools are very consistent in applying error checking. As shown in Sec. 5.1.2 above, these checks can result in the detection of a wide range of error types. In Sec. 4.2.2, we stated that static testing should catch all occurrences of module calling sequence errors, coding standards violations, and mixed-mode arithmetic. These correspond to the following error types from the TRW classification scheme:

- F_300 Subroutine arguments not consistent in type, units, order, etc.
- J_900 Software not compatible with project standards (only some of these will be caught)
- A_700 Precision loss due to mixed mode

However, static analysis has some limitations which keep it from detecting a high percentage of errors in all programs. These include:

- Only a very weak form of data flow analysis can be performed for arrays with variable subscripts. A static tool can consider the entire array to be one variable; if it does this, an assignment to any element "initializes" the array. Similarly, if any element of the array appears in an expression the array is considered "referenced".

- Static tools can only analyze structural aspects of a program's control flow. For example, a static tool will detect the dead code if an unlabeled statement follows an unconditional GOTO; but it cannot detect the problem if it is caused by the predicate of a branching statement (e.g. an "ELSE" clause following "IF (M > 0 OR M < 10)" is logically, but not structurally, dead code.)

Much more research must be done before we can associate error types from a classification scheme like TRW's with detection probabilities for static analysis. However, the error seeding experiment performed by Gannon, et al., at GRC indicates that static testing will not always catch the types of errors listed at the beginning of Sec. 5.1.2. Table 5.3 shows those error types that were seeded and the results for static testing with the SQLAB tool.

TABLE 5.3
RESULTS OF STATIC TESTING FOR THE TRW ERROR TYPES
OFTEN DETECTED BY STATIC ANALYSIS
(Source: Gannon, et al., Table 5.2, p. 5-5)

	Error Type	No. Caught	No. Missed
A_100	Incorrect operand in equation	0	2
A_400	Units or data conversion error	0	1
A_800	Missing computation	0	2
B_100	Incorrect operand in logical expression	0	1
B_200	Logic activities out of sequence	0	1
B_300	Wrong variable being checked	0	3
B_400	Missing logic or condition tests	2	1
B_600	Loop iterated incorrect number of times (including endless loop)	0	1
D_100	Data initialization not done	1	0
D_200	Data initialization done improperly	0	2
D_400	Variable referred to by wrong name	1	1
F_200	Call to subroutine not made or made in wrong place	0	2
G_100	Data not properly defined/dimensioned	1	1
H_300	Data units are incorrect	0	4

Program Characteristics

We believe that static testing is most effective when used early in the development of a program. The evidence is conflicting on this —the IMSL data from Howden, where 55% of the errors were detected by static methods, is for programs that had been in use for a substantial period of time. But if static analysis is used immediately after code is written, and after major revisions are made, it should screen out most data initialization errors and identify many of the grosser programming mistakes that were made.

The data compiled by Rubey, et al., support this contention. Their study compared static analysis with dynamic testing. They noted that static analysis caught most of those errors that were detected early (the first 40% of the validation phase), while dynamic testing caught most of the errors found during the remainder of the validation effort. They concluded that:

> "...the execution and (static) analysis methods are complementary; the analysis methods detect the earlier--perhaps easier--errors, while the execution methods continue to detect errors after the analysis methods are unproductive."[1]

It is no harder to test large programs with static analysis than it is to test single modules or small programs.[2] The documentation produced by static tools can be very useful in dynamic and formal testing of large programs. Program statistics and complexity measures can be computed from documentation reports to help plan other forms of testing.

Static analysis is useful for testing all types of programs: data flow, variable type compatibility, and interface errors are problems in programs of every application. Static testing does not provide information about timing characteristics, which are important to real-time and interactive programs. The general problem of deadlock in concurrent systems is not addressed very effectively by the structural deadlock detection capabilities of static tools.

[1]Rubey, et al., p. 153.

[2]Early static test tools required excessive amounts of computer time and storage to process large amounts of code. However, recently tools have been built which can easily accomodate most large programs. See subsection 3.1.4 on the computer resources required by static analysis.

Structured programming languages with strong data typing eliminate the need for some of the static checks described here. In effect, the compiler for these languages performs some of the functions of a static test tool. The static checks that are built into advanced languages such as PASCAL and ADA include:

- Module interface compatibility checking
- Coding standards enforcement
- Mixed-mode anomaly checking

In addition, most advanced languages permit recursive procedure calls.

There is no reason why more static checking cannot be included in compilers. As programmers become familiar with static analysis and recognize its value, they will want compilers that have static testing capabilities.

5.1.4 <u>Cost</u>

Static analysis requires a minimal amount of time from the user. The computer time and storage needed by static tools used to be a serious impediment to their use on large programs. However, now there are tools available whose resource requirements are linear functions of program size. Managers can make good estimates of their total static testing costs by looking at the computer costs for a few runs of the tool on code of similar size.

<u>Analysis Time</u>

Static testing requires a very small investment of time on the part of the user. There is little to learn about using a tool. The only code change that needs to be made before a static test can be run is to add assertions about variable usage and units, and this is optional. To test a piece of code, the user just selects the options and invokes the tool.

Some time is required to examine the output from a static test run. Error messages must be associated with mistakes in the code, and the user must decide whether each warning message from anomaly checking indicates a real error. However, static testing actually saves time for the tester, because it clearly identifies errors and helps to locate and correct them (see Chapter 6).

We found only one bit of data on the amount of analysis time required by static testing. In the GRC experiment with the missile trajectory program (5000 source lines), two person-hours were required to look over the output of the static tool and find the errors that it indicated in the program.[1]

Computer Resources

In Table 5.4 we show the time required to run several static tools on some different-sized programs. Notice that the time required per line of source is nearly constant for FAVS, while it increases with the size of the program for DAVE. The efficiency of FAVS was improved dramatically over an earlier version, which required an exponentially increasing amount of computer time to handle large programs.[2]

The developers of DAVE state that the tool runs in time that is "linearly proportional to the product of the number of edges in the flow graph (of the test program) and the number of program variables."[3] They have not had the opportunity to streamline their tool's operation. To do this, they must tailor the tool's data handling to the word length of

[1] C. Gannon, et al., op. cit., p. 1-11. The testers were professionals with 5-10 years of experience in the computer field, and were familiar with the tool being used.

[2] R. A. Melton, FAVS Enhancement Final Report, General Research Corporation CR-3-754/1, December 1980, p. 2-1.

[3] L. J. Osterweil and L. D. Fosdick, "DAVE - A Validation Error Detection and Documentation System for Fortran Programs", Software - Practice and Experience, Vol. 6, No. 4 (October-December 1976), p. 473.

TABLE 5.4

TIME OVERHEAD OF STATIC TOOLS

TOOL	COMPUTER	SIZE OF TEST PROGRAM (LINES)	COMPUTER TIME REQUIRED (CPU + I/O), seconds	
			TOTAL	PER LINE OF SOURCE
DAVE*	CDC CYBER 175	228	12.4	0.05
DAVE	CDC CYBER 175	500	46.2	0.09
DAVE	CDC CYBER 175	715	61.0	0.09
DAVE	CDC CYBER 175	704	76.0	0.1
DAVE	CDC CYBER 173	704	290.	0.4
DAVE	CDC 6400	228	57.9	0.25
DAVE	CDC 6400	965	592.	0.6
DAVE	CDC 6400	1579	2069.	1.3
FAVS†	CDC 6400	154	21.6	0.14
FAVS	CDC 6400	188	29.8	0.16
FAVS	CDC 6400	1612	223.1	0.14
FAVS	CDC 6400	1546	211.	0.14
FAVS	CDC 6400	3751	479.	0.13
FAVS	CDC 6400	7400	1011.	0.14
SQLAB§	CDC 7600	5000	24.	.0048
J73AVS**	AMDAHL 470	4000	—	.035

*K. A. Smith, op. cit., Table I.

†R. A. Melton, op. cit., p. 2-3.

§C. Gannon, et al., op. cit., p. 1-11.

**C. Gannon, R. F. Else, JOVIAL J73 Automated Verification System User's Manual (Preliminary Draft), General Research Corporation CR-4-947, July 1981, p. D-2.

a particular machine; the current version of DAVE is designed for maximum portability.

Unfortunately, we don't have data on the time required to compile the programs listed in Table 5.4. However, our experience with the FAVS tool has been that:

- It takes three to four times as much execution time to perform FAVS static analysis on a FORTRAN program as it does to run the same code through the FORTRAN compiler.

- For programs written in a structured FORTRAN dialect, it takes only two to three times as long to do FAVS static checking as it does to preprocess and compile the same code.

Smith[1] states that DAVE required from 25 times the compilation time (for 800 lines of code) to 130 times the compilation time (for 7200 lines) in tests on CDC machines.

Static tools require the following to be stored as permanent files:

- The tool itself
- Any data from previous static runs that can be used to test code that has been modified or added to a program

The second type of permanent file includes "stub module libraries" (see Sec. 5.1.1) and other data necessary to reconstruct an old test run. A tool may not be able to produce certain types of documentation from such a file. The way the library file works varies widely from tool to tool.

[1]K. A. Smith, Evaluation of Verification and Testing Tools for FORTRAN Programs, NASA Technical Memorandum 80205, July 1980, p. 3.

The temporary storage required to make a static test run includes the following:

- Enough core to permit the tool to execute
- Symbol tables for all variables and other operands
- Information about each executable statement
- The test program's calling tree and control flow graph

The symbol tables built by static tools are very similar to those used by compilers. The statement information is used in producing documentation, while the calling tree and flow graph are used in data flow and interface analysis. The sizes of these files depend upon the size and complexity of the program being analyzed and on the type of checking that the tool can perform.

Table 5.5 gives some data on the storage required by three static test tools in particular installations.

Static tools typically produce three kinds of output:

- A listing of the source code of the test program
- Error and warning diagnostic messages
- Documentation reports (at the user's option)

Occasionally, a static test run will produce an excessive amount of output. This is especially likely to happen when a large program is analyzed for the first time.

Procurement Costs

A version of the FACES tool sells for $1590. The static testing capability of RXVP80™ costs $14,000 (only $6000 if you already have the tool's controller). The documentation option of RXVP80™ costs an additional $4000.

TABLE 5.5

STORAGE REQUIREMENTS OF STATIC TOOLS

TOOL	COMPUTER	LOAD SIZE (Memory Words-Decimal)	TEMPORARY STORAGE	LIBRARY STORAGE (Words Per Statement)
FAVS*	UNIVAC 1100/40	50,000 (max. overlay)	Minimum 7 words per symbol, more as needed	About 1.5 times the size of the source file
J73AVS†	CDC 7600	65,000 (max. overlay)	150 words per source statement	50 words per source statement
DAVE§	CDC 6400	50,000 (max. per processing phase)	6500 words for 200-statement subprogram	§

*R. A. Melton, op. cit., p. 206 and pp. 3-1 through 3-2.

†C. Gannon and R. F. Else, op. cit., p. D-1. Both FAVS and J73AVS perform dynamic testing, and some of the storage overhead shown here is due to those preprocessing requirements.

§L. J. Osterweil and L. D. Fosdick, op. cit., p. 483. No data on the size of a permanent data base was found. DAVE does not have a stub library capability, and the developers do not recommend using the restart facility to analyze code changes.

Cost Worksheet

Since so little analysis time is involved in static testing the major cost of using static analysis is the computer cost.

It is a good idea to benchmark a static tool on a few different-sized programs when it is first obtained. This should be relatively inexpensive, and will give project managers a good indication of how much it will cost them to test their software.

A manager of a software project should plan to make several runs of the static tool on his code, since a new test has to be made each time code is revised.

5.2 EXECUTABLE ASSERTIONS

5.2.1 Summary of the Technique

Executable assertions are special statements inserted into the source code of a program. They allow the programmer to specify conditions that are required for correct operation of the program. If such a condition does not hold during execution of the program, this fact is reported via an error message. The programmer can also specify actions to be taken when an assertion is violated.

Most compilers do not recognize and translate assertions--an assertion preprocessing tool must be used. The tool generates code, in the same language as the rest of the program, which carries out the condition checking and error handling logic for the assertion. Different preprocessing tools recognize different forms of assertions. A programmer can augment a less powerful tool by writing code to do some of the condition checking.

There are assertion preprocessors available for most programming languages. However, most of the current tools lack some desirable features. Some work is being done to develop more advanced assertion preprocessors and to incorporate assertions into compilers.

Capabilities

Executable assertions are constructs added to a programming language. They do two things: indicate by an output message that something has gone wrong in a program, and permit the programmer to specify action that should be taken when such an error occurs. The general form of an executable assertion is:

```
ASSERT condition;
FAIL block;
```

Here "condition" is an expression that can be evaluated logically (as TRUE or FALSE) during execution of the program. The "fail block" is optional--it contains the error-handling code.

Assertions must be translated into executable code. This is usually done by a preprocessing tool, although some compilers will accept and translate assertions. The kinds of conditions that can be checked by assertions, and the syntax for declaring these conditions, vary from tool to tool. The types of assertions accepted by a tool are often referred to as its "assertion language".

The general form of a translated assertion is:

```
IF (NOT condition) THEN
     Print error message;
     Execute fail block
END IF
```

Executable assertions can do the following things:

- Indicate that a program is operating incorrectly
- Help the programmer to locate errors
- Indicate that a program is being used improperly
- Provide fault-tolerance in a program
- Express specifications and design intentions as in-line documentation of the program
- Form the basis for a formal verification of a program

Executable assertions can detect any error that can be expressed as a condition in the assertion language. Some important examples of such errors are:

- The result of a computation is outside of a range of reasonable values, or is inconsistent with another result.
- A variable does not behave as intended: it changes value when it should not, or it does not change in the desired way.

- Control flow is incorrect: the branch taken is incompatible with program conditions, or a special case is not handled properly.
- A call to a routine results in an unacceptable condition on return.
- The output of a routine is incorrect.

Assertion violations are reported with the location in the source program where the violation occurred. This can help isolate the location of the coding error that caused the violation. We discuss how assertions are used in debugging in Sec. 6.1.

Assertions can also guard against an improper use of a program or routine that is otherwise correct. This is done by imposing conditions on the inputs to the code. Yau, et al.[1] describe how assertions can be used to protect data structures from misuse or accidental destruction.

Andrews[2] discusses how the "fail block" feature can be used to recover from error conditions or to provide "graceful degradation" in a program's performance. We feel that fault tolerance is an important feature of executable assertions, but this topic is beyond the scope of this report.

[1]S. S. Yau, J. L. Ramey, R. A. Nicholl, "Assertion Techniques for Dynamic Monitoring of Linear List Data Structures", The Journal of Systems and Software, Vol. 1 (1980), pp. 319-336.

[2]D. M. Andrews, "Software Fault Tolerance Through Executable Assertions," Proceedings - Twelfth Annual Asilomar Conference on Circuits, Systems, and Computers, November 1978, Pacific Grove, California.

The presence of assertions in program code is a form of documentation. Assertions are useful when code is being modified, since they remind the programmer of conditions which must hold for the program to operate correctly. Sometimes specifications for a program can be stated directly as assertions.

Formal techniques use assertions to prove the correctness of a program (see Sec. 5.5). To successfully verify a program, a more extensive set of assertions is needed than what will typically be used in a testing effort. Assertions written for testing can be used in a verification effort, and the assertions generated during program proving have all the properties of executable assertions discussed here.

Operation

The user must decide what assertion checks to make, encode them in the assertion language, and insert them in the code. Assertions should be developed and "programmed" at the same time as the code itself, for several reasons:

- Writing the assertions increases the programmer's understanding of the purpose and design of the program.
- The assertions themselves will have mistakes which have to be debugged.
- Assertions are useful throughout the life of the program.
- Adding a full set of assertions to a large already coded program is a tedious job that no one will want to do.

Different assertion languages permit different conditions to be asserted. However, a programmer can write code in the source language to get around the shortcomings of his preprocessing tool. Some useful constructs which have been included in assertion languages include:

- ASSERT (exp), where "exp" is any legal logical (boolean) valued expression in the source language, including a function

- A form of the first-order predicate calculus which checks multi-element data structures. These constructs look like:

  ```
  ASSERT (ALL i IN range) condition
  ASSERT (SOME i IN range) condition
  ```

  where "i" is a dummy index to the data structure, "range" describes the limits on the index over which the assertion is to hold, and "condition" is any legal condition in the assertion language involving the indexed elements of the data structure.

- Notation which refers to previous values of variables. In this way, iterations can be checked to see that the progression of values for the same variable is correct. The assertion language may provide shorthand such as:

  ```
  OLD (var,num),
  ```

  where "var" is the name of a variable, and "num" indicates which previous value is desired (1 means the last, 2 the one before that, etc.).

  This may be accomplished in more primitive assertion languages by adding special variables.

- Notation which specifies flow of control. Chow[1] has suggested that assertions could check the order that variables are defined and referenced on a path with constructs like:

[1]T. S. Chow, "A Generalized Assertion Language," Proceedings - 2nd International Conference on Software Engineering, October 1976, San Francisco, p. 395.

PATH IS (REF var; DEF var),

where "var" is the name of a variable.

This kind of checking can be done with simpler assertion tools by using a set of flags or counter variables.

Automated Tool Support

Tools which process executable assertions exist for most common high-order languages. Table 5.6 lists those covered in the tools survey and gives the languages and computer systems for which they operate. None of these tools accept all of the types of constructs described in the last subsection. Taylor[1] describes a more elaborate preprocessor which is being developed for HAL/S. ADA has an assertion construct equivalent to the first type listed above.

5.2.2 Effectiveness

Assertions can be used to detect a wide variety of errors. They are most effective against computation errors, and have been shown to catch high percentages of data handling and logic errors, too. However, it is more difficult to write assertions that catch these last two types of errors.

[1] R. N. Taylor, "Assertions in Programming Languages", SIGPLAN Notices, Vol. 15, No.1 (January 1980), pp. 105-114.

TABLE 5.6

EXECUTABLE ASSERTION PREPROCESSING TOOLS

Source: Sec. 2 of First Interim Report

Tool	Languages	Computers
ACES	FORTRAN	CDC, IBM
JAVS	JOVIAL J3	CDC 6400, HIS 6080/6180
J73AVS	JOVIAL J73	ITEM AS/5, DEC20
PET	FORTRAN	CDC 6000/7000, IBM 360/370 OS, UNIVAC 1100
SQLAB	FORTRAN, PASCAL, JOVIAL J3B	CDC 6400/7600
TPL	FORTRAN	?
V-IFTRAN™	FORTRAN	Any system supporting FORTRAN

Only a small amount of data is available on the percentage of program errors detected by assertions. The studies done at GRC have indicated that an extensive set of assertions can catch more than 70% of total errors.

Types of Errors Detected

Assertions detect violations of conditions that must hold during the execution of a program; hence, they deal almost exclusively with symptoms of errors. There is only one error category in the TRW scheme which describes a condition which can almost always be tested for with an assertion: "G_200 Data referenced out of bounds." However, studies of assertion testing have shown that many errors from the following major categories can be caught:

- A_000 Computation Errors
- B_000 Logic Errors
- C_000 Data Input Errors
- D_000 Data Handling Errors
- F_000 Interface Errors
- G_000 Data Definition Errors
- H_000 Data Base Errors

The rest of the major error categories usually cannot be addressed by executable assertions because they involve things that are external to the program itself. For example, data output errors (E_000) cannot be detected if the output devices and the routines that control them are separate from the test program (as is usually the case).

Benson and Saib[1] showed how assertions can be used to detect computational, logic, and data handling errors. They seeded three errors of each type into a test program and formulated assertions to detect the errors. Their conclusions were:[2]

- "Assertions are most valuable for catching computational errors. These errors can be found by specifying variable ranges and by stating approximate bounds on the results of computations."

- "Data handling errors can usually be detected by assertions which specify ranges, units, scale factors,..." They felt that static analysis is usually better at this, however.

[1]J. P. Benson and S. H. Saib, "A Software Quality Assurance Experiment", Proceedings of the Software Quality and Assurance Workshop, San Diego, November 1978, pp. 87-91.

[2]Ibid., pages 88 and 90.

- "Logic and sequencing errors are the most difficult to detect using assertions."

Two experiments have been conducted at GRC in which errors from the TRW scheme have been seeded into test programs.[1] Since each experiment used only a small number of errors, the sample sizes for each error category are hardly statistically significant. However, the two sets of results are consistent with each other and show assertions to be most effective for computational errors. Table 5.7 compares the results of the two experiments for those major error categories which were seeded in both tests.

TABLE 5.7

PERCENTAGES OF ERRORS DETECTED IN SELECTED MAJOR ERROR CATEGORIES DURING ASSERTION TESTING EXPERIMENTS

		Percent of Errors Detected	
	Major Error Category	Benson and Andrews	Test 2
A_000	Computation Errors	80%	83%
B_000	Logic Errors	60%	73%
D_000	Data Handling Errors	78%	80%

[1]The first experiment is described in J. P. Benson and D. M. Andrews, Adaptive Search Techniques Applied to Software Testing, General Research Corporation CR-1-925, February 1980. The other one is the Test 2 results described in Appendix D.

Error Detection Ratio Data

We can cite three results on the percentage of errors detected during experiments with executable assertions. Unfortunately, there is no data from live experience with assertion testing.

- The first set of assertions developed by Benson and Andrews detected nine of 24 seeded errors, or 38%. By adding more assertions, they were able to detect eight more errors, for a total of 71%.[1]

- The results of Test 1 of the assertion experiment described in Appendix D were 21 out of 24 errors detected, or 87%. In this experiment, the errors were known when the assertions were developed.

- The set of assertions used in Test 1 was tested for a different set of seeded errors in Test 2. This time 25 of 34 errors were caught, or 74%.

5.2.3 Reliability

How reliable assertions are depends upon the person writing them. To write good assertions, a programmer must understand the way his program is supposed to operate, be familiar with the assertion language, and be thorough in his use of assertions. Assertions have to be debugged just like the rest of a program.

The test data used has a great effect on the reliability of executable-assertion testing. Test data must cause assertions to be violated or errors will go undetected. To be effective, assertion testing should be combined with a systematic method of generating test data, such as structural or functional testing.

[1]J. P. Benson, D. M. Andrews, ibid., p. 1-9.

Assertions are an imperfect method of error detection for two main reasons: not all error conditions can be described in assertions, and sometimes only weak conditions can be imposed.

Assertions are useful throughout the software life cycle and on programs of all sizes. More research needs to be done on the use of assertions in specific application areas.

Human Factors

Writing assertions is a creative activity. We have proposed some guidelines on where assertions should be located and what kinds of conditions they should check for (see Sec. 7.2 and Appendix D). However, the final responsibility for developing an effective set of assertions lies with the user.

To write assertions, a programmer needs to understand how the program is designed and coded. He also needs to develop some skill in handling assertion constructs--this comes with a little experience. To write good assertions, a programmer must then do the following:

- He must find out enough about the application area of the program to develop tight bounds on the values of variables and the results of computations.

- He must write assertions which can trap special error conditions such as logic and data flow errors. This can be difficult when using an assertion language of limited power.

- He must be thorough--all conditions that can be checked for assertions must be identified.

Assertions have to be debugged after they are put in a program. Assertions are susceptable to many programming errors, including:

- The asserted condition is misstated.
- The assertion is in the wrong place.
- A computation or value needed in the assertion is omitted.

The input data used to test a program affects the reliability of executable-assertion testing. Assertions won't be violated if they aren't exercised. When they are exercised, they won't be violated under all conditions. We do not discuss test data generation methods in this section, since executable assertions can be used regardless of the way test data is chosen. But as an indication of how important test data is to the effectiveness of assertions, we note that 43% of the undetected errors in Benson and Andrews' experiment might have been detected if more tests had been run.[1]

Kinds of Errors

The error conditions checked by assertions must be stated in the assertion language. A fairly wide range of errors can be addressed by assertions, but the constructs have their limitations. For example, no assertion language permits statements like "X IS INITIALIZED"; at run time, the variable X has a value, whether by accident or design. In general, assertion conditions must describe symptoms of errors which can be expressed using information available within the executing program.

It is often necessary in an assertion to use a rather loose check on the value of a variable or result of a computation. Such an assertion will allow some errors to go undetected. For example, it may be possible to check the results of a numerical algorithm only to within a few orders of magnitude of the desired precision without using the same algorithm to produce the check value. In such cases, assertions can only act as a filter for some programming mistakes—the accuracy of the results must be verified by other means.

[1]This is based on an examination of Table 6.1, p. 6-2 of Benson and Andrews.

In Table 5.8 we show the error categories used in the two error seeding experiments performed at GRC (Benson and Andrews and the Test 2 experiment from Appendix D. The numbers of errors detected and missed are tallied for each category. More studies should be performed to make statistically significant sample sizes available in each category.

Program Characteristics

Assertions are useful throughout the lifecycle of a program. They force the programmer to consider the program's purpose and specifications as he codes. They help to detect and locate errors during testing and verify corrections during retesting. The documentation supplied by assertions is useful during maintenance.

Assertions can be used successfully with programs of any size. Of course, larger programs require more assertions for effective testing.

The relationship between the size and complexity of a program and the number of assertions that should be used to test it was explored in the assertion testing experiment described in Appendix D. The widely used Halstead and McCabe metrics were not found to be good indicators of the number of assertions needed.

Research should be done on the use of assertions in programs of various applications and in real-time and systems programs. There may be new assertion constructs that would be useful in special situations, such as describing timing conditions in concurrent programs.

5.2.4 Cost

The costs of assertion testing depend on the number of assertions placed in the code, the difficulty of writing and debugging them, and the number of test runs made with the asserted program. There is little data or experience that can be used to gauge the magnitude of these costs. Writing and debugging assertions can be expected to add signi

TABLE 5.8
RESULTS OF ERROR SEEDING EXPERIMENTS FOR
EXECUTABLE ASSERTION TESTING

	Error Type	No. Caught	No. Missed
A_100	Incorrect operand in equation	2	1
A_200	Incorrect use of parenthesis	1	0
A_300	Sign convention error	3	0
A_500	Computation produces an over/under flow	1	0
A_600	Incorrect/inaccurate equation used	1	1
A_800	Missing computation	1	0
B_100	Incorrect operand in logical expression	0	1
B_200	Logic activities out of sequence	3	1
B_300	Wrong variable being checked	1	2
B_400	Missing logic or condition tests	5	1
B_500	Too many/few statements in loop	1	0
B_700	Duplicate logic	1	0
C_100	Invalid input read from correct data file	2	0
D_100	Data initialization not done	2	2
D_200	Data initialization done improperly	1	0
D_300	Variable used as a flag or index not set properly	1	0
D_400	Variable referred by the wrong name	3	0
D_500	Bit manipulation done incorrectly	1	0
D_600	Incorrect variable type	3	1
F_100	Wrong subroutine called	1	1
F_200	Call to subroutine not made or made in wrong place	1	0
F_700	Software/software interface error	1	1
G_100	Data not properly defined/dimensioned	1	1
G_300	Data being referenced at incorrect location	1	1
H_100	Data not initialized in data base	2	0
H_200	Data initialized to correct value	2	1
H_300	Data units are incorrect	0	1

ficantly to costs at the beginning of a project, while the overhead of making test runs should not be as great. More research on the use of assertions in program development is badly needed.

Analysis Time

Developing an effective set of assertions for a program requires a considerable investment of time. The size of the job depends on the number of assertions needed to check for possible error conditions and the difficulty of formulating them. Presently there is no measure of how "well-asserted" a program is; it would be very nice to have such a measure, to use in test planning and to guide formulation of assertions.

Unfortunately, we also have no data on the analysis time for assertion testing of a program. In the absence of such information, we postulate that the time required to write a given number of assertions is on the order of the time to write the same number of lines of source code. We have already commented on the similarity of the tasks of writing a program and developing the assertions for it.

The other major task associated with executable assertion testing is examining the assertion violations to determine the program errors that caused them. However, this does not represent analysis time beyond that required by conventional testing, because the same test runs would have to be debugged without the information provided by the assertion violations. Assertions save time (once they are debugged) because they help to locate errors as well as to detect them.

Computer Resources

Like the analysis cost, the computer resources used in assertion testing depend upon how thoroughly assertions are used. The categories of computer overhead are:

- The time required by the preprocessor and compiler to turn the assertion into executable code

- The extra execution time required by the assertion checks
- The extra space required by the source and executable versions of the program due to the assertions

The preprocessor and compiler overhead will be incurred each time either the assertions or the code are changed. The execution time overhead will be incurred once for each test run. Once a production versi^-- of the program is achieved, the assertions can be removed by recompiling the program with the assertions disabled. In this way, the execution time overhead is not incurred by the end-user of the program; but he also does not have the protection that the assertion checking could provide him.

We have some data on the overhead for some assertion testing experiments. The experiments are described here, and the overhead data is presented in Table 5.9.

TABLE 5.9
COMPUTER OVERHEAD OF ASSERTION TESTING

	Percent Increase In		
Source	Compile Time	Execution Time	Storage Space
SQLAB code	5.5%	4%	8%
Radar Simulation	56%	12%	13.5%
Yau, et al. (4 programs)	55-125%	0-40%	6-12%

- The SQLAB tool was used to preprocess assertions placed in its own code.[1] Assertions were added for about 40% of the decision statements, or roughly one assertion for every ten executable statements. Few fail blocks were included.

- Benson and Saib used a 1000-line radar simulation to demonstrate the use of assertions in providing fault-tolerance. All decision statements had assertions and fail blocks, for a ratio of one assertion to every five executable statements.[2] SQLAB was the preprocessing tool.

- Yau, et al.[3] tested four JOVIAL programs of roughly equal load size (17000 words). They used the JAVS tool to preprocess the assertions. They give no data on the density of assertions in their paper.

Occasionally, an assertion test will produce an excessive amount of output. This is most likely to happen when testing a program with a loop in it—if an assertion is violated every time through the loop, this may mean a lot of violation messages. Some tools allow the user to write his own error message routine; a limit on the number of violation messages can be imposed in that case.

Procurement Costs

The V-IFTRAN™ tool, which preprocesses assertions and structured control constructs for FORTRAN, sells for $6370.

[1]Saib, et al., op. cit., p. 170.
[2]Andrews, et al., op. cit., pp. 334-335.
[3]Yau, et al., op. cit., pp. 334-335.

Cost Worksheet

The costs of executable assertion testing are summarized in the formula:

Cost of writing and debugging assertions

+

Overhead per test run x Number of test runs

The best available estimate of the cost in analyst time to develop the assertions is the cost to write an equivalent amount of code. For the GRC experiments cited in Table 3.9, this represents ten to twenty percent overhead in coding and debugging costs. Studies of real development projects should be made to refine this estimate.

The compile time overhead tends to be higher than the execution overhead for assertions; so testing is most efficient if the assertions are debugged early and as many program errors as possible are detected between code changes. The number of test runs depends on the test data generation method used.

5.3 STRUCTURAL TESTING

5.3.1 Summary of the Technique

Structural testing (also known as branch or path testing) describes a goal rather than a method of testing. The goal is to increase the amount of code tested. It is impossible to test all the paths or combinations of branches in a large program. It is possible to test all branches. For effective testing, all branches and as many paths as possible should be tested.

Instrumentation tools are used to determine how much coverage is achieved in a test run. These tools can also provide timing data, execution traces, and other information. However, the tester himself must formulate input data and decide whether the program has run correctly for each test.

Many instrumentation tools have been developed in the last ten years. Tools are available for most programming languages and computers.

Capabilities

The purpose of structural testing is to ensure that a program is thoroughly exercised. Several measures of thoroughness, or test coverage, can be used to quantify this goal. The measures are based on structural units of the source code of the program being tested—hence the term "structural" testing. The structural units we will consider are:

- Statements, which in this context means executable statements
- Branches, which correspond to the outcomes of each decision statement in a program

- Combinations of branches (this is similar to the Linear Code Sequence and Jump (LCSAJ) metric defined by Woodward, et al.[1]); and
- Paths, which correspond to distinct ways of executing a program from entry to exit.

For all but very simple programs, execution of all program paths or of all possible combinations of branches is impossible. Coverage of all branches, which implies execution of all statements, is a reasonable goal for large programs. However, there are many errors that will go undetected if attention is not paid to the combinations of branches which are executed. So in a sense structural testing is open-ended: all branches and as many different paths as possible should be executed.

Automated tools can be used to tell how much coverage has been achieved in a test run. The program must first be run through a preprocessor, which inserts code that collects the coverage information during execution. This is usually called "instrumenting" the code. Structural testing is an iterative process; the user compares the amount of coverage achieved to his goal, then tries to formulate new test data to increase the coverage.

The code that collects coverage data can collect other information as well. The nature of this information depends on the tool used and the level at which the code is instrumented. If probes are inserted after every statement in the program, then the entire history of the execution of a program can be recorded. Of course, instrumenting at the statement level will incur significant computer overhead. To determine

[1]M. Woodward, D. Hedley, and M. Hennell, "Experience with Path Analysis and Testing of Programs," IEEE Transactions on Software Engineering, Vol. SE-6, No. 3 (May 1980), pp. 278-286.

branch coverage, it is only necessary to insert probes at every decision statement.

Structural testing, and using an instrumentation tool, can provide the tester with the following information:

- It can reveal untested parts of a program, so that new test efforts can be concentrated there.

- Data on the frequency of execution of parts of a program, and the time required to execute them, can be tabulated. This information can be used to make the program more efficient.

- The range of values assumed by a variable (high, low, average, first, last) can be recorded and checked for reasonableness.

- A trace of what has occurred at each statement in a section of code can be printed. This can be useful when debugging.

- The data flow patterns of variables can be analyzed from the execution trace file.[1] In this way errors and anomalies in the use of subscripted variables can be detected.

Operation

An instrumentation tool requires the test program source code as input. Usually another file in the source language is produced that has the probes in it. The tool also needs information about the control structure of the program—if the tool also performs a static analysis of

[1]J. C. Huang, "Detection of Data Flow Anomaly Through Program Instrumentation", IEEE Transactions on Software Engineering, Vol. SE-5, No. 3 (May 1979), pp. 226-235.

the program it can use the tables produced there. Most tools allow the user to control the level of instrumentation and to designate what parts of the program are to be instrumented.

There is no automated mechanism for error detection in structural testing. The user must recognize errors by looking at the output of the program. We describe how other test techniques can be used to help detect and locate errors during structural testing in Chapter 6 and in Sec. 7.3.

When a user finds errors in his program during the course of testing and makes code changes to correct them, the new version of the program must be run through the instrumentation tool again.

<u>Automated Tool Support</u>

There are a lot of instrumentation tools available. They accommodate most major high-order languages and many computer systems. Table 3.10 lists the tools surveyed in the first interim report. Most of the tools do not have all of the capabilities described above—however, all can at least provide coverage information.

There are automated tools associated with other test techniques that can help generate test data for structural testing. Some static tools provide "reaching set" information, which gives the branch conditions along the paths to a statement. Symbolic execution extends this approach by attempting to form an algebraic expression for the branch conditions in terms of input variables. Unfortunately, these methods have not proven very successful in practice; they do not often simplify the job of generating test data for complicated programs.

5.3.2 <u>Effectiveness</u>

When test data is carefully chosen, structural testing can be very effective. TRW determined that a combination of structural and functional testing could detect a high percentage of many types of errors from the Project 3 study.

TABLE 5.10
CAPABILITIES OF INSTRUMENTATION TOOLS
Source: Tools Survey, Appendix B

TOOL	LANGUAGES	TIMING DATA	EXECUTION TRACE	VARIABLE RANGES
ACES	FORTRAN	✓	✓	
CAVS	COBOL			
FAVS	FORTRAN			
FORTRAN Analyzer	FORTRAN			
Instrumenters I and II	FORTRAN			
ISMS	ALGOL 60	✓	✓	
JAVS	JOVIAL J3	✓	✓	
J73AVS	JOVIAL J73	✓	✓	
NODAL	FORTRAN			
OPTIMIZER III	COBOL	✓	✓	
PACE	FORTRAN			
PET	FORTRAN	✓	✓	✓
PRUFSTAND	SPL (PL/1-based)		✓	
RXVP80™	FORTRAN			
SADAT	FORTRAN			
SQLAB	FORTRAN, PASCAL JOVIAL J3B-2			
TAP	FORTRAN			✓
Test coverage Code Auditor	JOVIAL J73/1			
TPL	FORTRAN			
V-IFTRAN™	FORTRAN			

The available data on the percentage of program errors detected by structural testing varies over a wide range. One branch testing study determined that 21% of a sample of errors could be detected; another study concluded that structural testing could catch 92% of the errors in a different set of programs. Structural testing has been shown to be effective for detecting errors early in the development life cycle.

Types of Errors Detected

The TRW error categories are not very useful for analyzing the effectiveness of structural testing. However, the TRW report examined the effectiveness of a combination of structural and functional testing for the errors in the Project 3 study. Their results can be considered a sort of upper bound on the effectiveness of structural testing--what happens when coverage is driven up by input data that is most likely to catch errors. The percentage of errors from each major category that TRW estimated could be detected this way is shown in Table 5.11.

TABLE 5.11
ERROR TYPES DETECTED BY STRUCTURAL AND FUNCTIONAL TESTING
Source: Table 4-32 of the TRW Report

Error Category	Percent Detected
Computational	87.4%
Logic	79.6%
I/O	98.5%
Data Handling	76.9%
Operating System/Support Software	100%
Routine/System Software Interface	27.3%
Tape Processing Interface	100%
User Interface	89.6%
Data Base Interface	100%
Global Variable/Compool Definition	87.5%
Documentation	64.7%
Requirements Compliance	75%

Error Detection Ratio Data

We found five pieces of data on the percentage of errors detected in various programs by structural testing:

- Howden[1] studied six programs written in different languages. Of 28 total errors, he determined that branch coverage testing would detect 6, or 21%. Complete path coverage could detect 18 errors, or 64%.

- Howden also studied the effectiveness of structural testing for the errors in the IMSL programs.[2] He determined that branch testing could detect 13 of the 42 errors considered, or 31%. Path testing detected an additional 3 errors, for a total of 38%.

- Mangold[3] determined that structural testing could detect 206 of a set of 224 errors, or 92%.

- Gannon, et al.[4] used branch testing on the eight Kernighan and Plauger programs that we discuss in Appendix D. They found 18 of the 26 errors, or 69%.

- TRW estimated that structural and functional testing combined could detect 72.9% of the errors in the Project 3 study.[5]

[1] W. E. Howden, "An Evaluation of the Effectiveness of Symbolic Testing", Software--Practice and Experience, Vol. 8 (1978), p. 387.

[2] W. E. Howden, "Functional Program Testing", IEEE Transactions on Software Engineering, Vol. SE-6, No.2 (March 1980), p. 167. Howden evidently considered a different set of errors here from the IMSL study cited on p. 5-11.

[3] E. R. Mangold, "Software Error Analysis and Software Policy Implications", IEEE EASCON, 1974, pp. 123-127.

[4] Gannon, et al., op. cit., p. 2-2.

[5] Table 4-33, p. 4-162 of the TRW Report.

Deutsch[1] provides other evidence of the effectiveness of structural testing. After the traditional checkout of the separate units of a large program (170,000 lines of code, 400 units or threads), the RXVP80™ tool was used to get complete branch coverage of each unit. Some 400 additional errors were discovered in this way. His paper gives no information about the percentage of total program errors that this constituted.

5.3.3 Reliability

The test data used has a great deal of influence on the reliability of structural testing. Input data that tests boundary conditions or singularities and demonstrates the operation of program functions should be used when doing structural testing.

Structural testing is guaranteed to find errors only when a program path handles all input data incorrectly. Since this is not the case for all errors, structural testing alone cannot ensure that a program is operating correctly.

Human Factors

A great deal of skill and judgment is required to formulate test data for structural testing. Because test data generation tools are inadequate, the tester must rely on his knowledge of the program to find ways to execute previously unexplored areas of code. It is also very important that each set of test data be designed so that program errors are revealed. Gannon, et al. emphasize that "(structural) testing should always be coupled with stress or boundary condition testing."[2]

[1]M. S. Deutsch, "Software Project Verification and Validation", Computer, April 1981, pp. 66-67.

[2]Gannon, et al., op. cit., p. 1-12.

Instrumentation tools give good feedback on the amount of coverage achieved by each test case. This documents the thoroughness of structural testing, and ensures that full statement or branch coverage is in fact achieved. The user must judge the thoroughness of path or LCSAJ testing of large programs because complete coverage is impossible.

Types of Errors Detected

Howden[1] defined four types of programming errors and looked at how reliable structural testing is for each of them. His approach is quite theoretical--he defines a "reliable test" and proves theorems about reliability. He treats only the case of complete path coverage, since this is the easiest to handle mathematically.

Even though they are not based on "practical" experience, Howden's results provide a lot of insight into how structural testing operates. The essence of most of his arguments is that every input that causes a particular path to be executed must be handled incorrectly by a program in order for it to be guaranteed that structural testing will catch the error. We briefly summarize his findings for each error type:

- Computation errors: structural testing will catch many of these errors, but it is impossible in general to tell how many.

- Domain errors: these occur when an error in a decision statement causes some inputs to be handled by the wrong part of the program. Structural testing is guaranteed to catch this error type only if the part of the program handling the incorrect cases no longer handles any of the ones it is

[1] W. E. Howden, "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-2, No.3 (Sept. 1976), pp. 208-215.

supposed to. For example, using "GT" instead of "LE" in an IF statement will be detected; but if "GE" was intended the error may not be found.

- Subcase errors: these occur when a program fails to distinguish a subcase of the input data, instead handling it like other data which is processed correctly. Structural testing is not reliable for this type of error—there is no guarantee that test data which belongs to the incorrectly handled subcase will be chosen to execute the path.

- Combinations of errors: errors that structural testing can detect singly may not be detected when other errors are present. This can happen if errors "mask" each other for some inputs. Errors may change the nature of the paths in the program and the inputs to each path—this can make the reliability results for single errors invalid.

Program Characteristics

It is best to do structural testing after some initial coverage is obtained by other testing methods. It is easier to achieve thorough coverage for small amounts of code, so structural testing should be used at the module or unit testing level if possible. Structural testing of large and complex programs is difficult, but these are the programs that most need thorough test coverage.

Good design and coding techniques can do a lot to make structural testing easier. Structured programming makes test data generation easier, since the control flow is easier to read and understand.

Structural testing can be used with programs of any type of application. The timing information provided by instrumentation tools is useful in improving the efficiency of time-critical routines.

5.3.4 Cost

The most expensive part of structural testing is the analysis needed to develop test data and examine output for errors. We need more experience with structural testing before we can develop good estimates of analysis time and cost. Available data suggest that using structural testing to debug programs requires 0.5 to 2.0 person-days per error found.

There is a good deal of data available on the computer overhead of instrumentation tools. The amount of overhead depends on several factors, including the level of instrumentation and the options selected. Generally, instrumentation tools require:

- A 20 - 100% increase in program size
- A 2 - 50% increase in execution time

There are a lot of commercially available instrumentation tools, most of which sell for less than $10,000.

It is hard to estimate the total costs of structural testing, because no one knows how to estimate the analysis time or number of test runs required. A user of structural testing on a large software project claimed a significant cost saving over traditional testing methods.

Analysis Time

A significant amount of time is required to do structural testing. In each test iteration, input data must be developed and the results of the test run must be examined for errors. There is currently no way to estimate the number of test runs needed to achieve a given level of coverage in all programs. Zeil and White[1] have obtained some nice results for programs with linear branch conditions, but most structural testing to date has proceeded by trial and error.

[1]Gannon, et al., op. cit., Table 1.3, p. 1-11.

Gannon, et al., used branch testing to debug (detect and locate) errors in a 5000-line trajectory analysis program. Errors were seeded into the program one at a time, and the set of tests which provided the branch coverage was run. An average of three person-hours of analysis was required for each error found.[1]

Deutsch reported that it required about two person-days per error for the 400 errors found in his use of branch testing. This includes the time to formulate the test cases and the debugging time.[2]

Computer Resources

The categories of computer overhead for instrumentation tools are similar to those for assertion tools; preprocessing and compile time, execution time, and program storage. The way the overhead is incurred is similar too—one preprocessing run is needed per version of the test program, and execution overhead is incurred on each test run. One difference between assertions and instrumentation is that instrumentation tools must spend some time processing the data collected at the end of a test run.

The overhead of instrumenting a program depends on:

- The tool used
- The level at which the code is instrumented
- The options selected—execution and variable value traces take lots of storage and time
- The structure of the test program—more branches means more probes

[1]Gannon, et al., op. cit., Table 1.3, p. 1-11.
[2]Deutsch, op. cit., p. 67.

We have data on the overhead required by several instrumentation tools:

- JAVS instrumentation doubles the size of the source and executable files, and increases execution time by 50%. Instrumenting and compiling the source code requires .09 CP hours per 1000 statements on an HIS 6180 computer.[1]

- J73AVS produces instrumented programs that are 1.5 to 2 times the original source file size. Execution time is increased by 50%. Instrumentation requires 4 to 8 CP seconds per 1000 statements on an ITEL AS/5.[2]

- Two sets of data are available on NODAL. Used to test a very large program on an IBM 360 (177 routines, 254K bytes of core per overlay), the tool increased execution time by 7% and core storage by 30%. On a somewhat smaller program on a UNIVAC 1108 (61 routines, 40K words per overlay), it increased execution time by 2% and storage by 20%.[3]

- Tests of PET on a CDC Cyber 173 gave the following results: execution time increased by 20 - 50%, storage required by executable code increased by about 30%.[4]

- The developers of ACES report a 20% increase in both execution time and storage overhead for their tool.[5]

---

[1]C. Gannon, N. B. Brooks, JAVS - Jovial Automated Verification System, General Research Corporation CR-1-722/1, June 1978, pp. E-2 and E-3.

[2]C. Gannon and R. F. Else, op. cit., pp. D-1 and D-2.

[3]Viewgraph copies distributed by Mr. Richard Maitlen, TRW Applied Software Laboratory, Redondo Beach, CA.

[4]K. A. Smith, op. cit., p. 6.

[5]C. V. Ramamoorthy and S. F. Ho, "Testing Large Software with Automated Software Evaluation Systems", in Current Trends in Programming Methodology, Volume II (R. T. Yeh, ed.), Prentice-Hall, 1977, p. 146.

Most of the output produced by instrumentation tools takes the form of concise reports. Information on coverage, timing, values of variables, and data flow can be presented in a few pages. However, execution traces can produce tons of output. They should be used with care. To avoid unnecessary overhead during other phases of testing, an uninstrumented version of the test program should be used once structural testing is finished.

<u>Procurement Costs</u>

Here are the costs of the commercially available instrumentation tools from the tools survey in the first interim report:

- INSTRUMENTERS I & II—$5000 each, permanent license.
- OPTIMIZER III (also has special debugging features)--$9750 to $28,500 depending on options.
- RXVP80™--$16,000 (only $8000 if you already have the static option or controller).
- V-IFTRAN™ (also preprocesses assertions and structured FORTRAN constructs)--$6370.

A survey of software tools performed by the National Bureau of Standards in 1977 reported the typical price range of instrumentation tools to be $2000 to $6000.[1]

<u>Cost Worksheet</u>

The total costs of structural testing are summarized by:

[1] I. T. Hardy, B. Leong-Hong, and D. W. Fife, <u>Computer Science & Technology: Software Tools: A Building Block Approach</u>, Nat'l Bureau of Standards Special Publication 500-14, August 1977, p. 8.

Computer overhead per test run x Number of test runs

+

Cost of analysis involved in generating test data and examining test output

The computer overhead for a test run can be estimated fairly accurately from the type of information given in the "Computer Resources" subsection above. However, it is difficult to estimate the number of test runs needed.

The major component of testing expenses is likely to be analyst costs. More experience with structural testing is needed to develop a way to estimate this.

Deutsch claims an overall saving in testing costs from using structural testing.[1] The money saved is due to earlier detection of errors in the development life cycle compared to traditional testing methods. The estimated net saving is over five times the cost of structural testing.

[1]Deutsch, op. cit., p. 67.

5.4 FUNCTIONAL TESTING

5.4.1 Summary of the Technique

Functional testing means generating test data based on knowledge of the functions performed by the test program and of the nature of its inputs. A large number of test cases can be generated this way for most programs. Thus, functional testing is open-ended; there are no metrics to indicate the thoroughness of testing or to tell when testing can stop.

Functional testing is supported by two types of automated tools--test harnesses and stress testing tools. Test harnesses make it possible to test partially completed programs, and to test large programs with complicated external interfaces. Test harnesses also have special debugging features--these are described in Sec. 6.3.

Stress testing tools use mathematical optimization techniques to automatically generate test data. The objective of stress testing is to find test points where a program exhibits undesirable behavior.

Capabilities and Operation

Functional testing is really *traditional testing*[1]--the objective is to generate test data that will find errors in a program. Functional testing is sometimes referred to as "black box" testing, because detailed information about the program's internal structure need not be used to formulate the test data. Instead, test data is chosen in the following ways:

[1]Howden made the term "functional testing" prominent in his seminal paper "Functional Program Testing" (op. cit.). Elsewhere (p. 2 of "A Survey of Dynamic Analysis Methods", in Effectiveness of Program Validation Methods for Scientific Programs) he has admitted that "The traditional requirements-based program testing method is functional testing."

- Data is chosen to explore whether the program correctly performs the functions that it is intended to perform. The functions should be described in the requirements and specifications for the program.

- The inputs to the program are examined. Using knowledge of the quantities they represent and how the program functions ought to operate on them, the set of possible values for each input variable can be partitioned into "subdomains". Test data sets are generated by taking combinations of samples from each subdomain.

- Some measure of the program's output behavior is defined. Test data is sought which drives this measure toward an undesirable value. Techniques from mathematical optimization can be used to do this.

Sometimes in the literature these test data generation methods are considered separately. The input subdomain method is also referred to as "special values testing"; the last method is sometimes called "stress testing".

Like structural testing, functional testing has no automatic method of error detection; errors are found by manually examining the test program's output. To an even greater degree than structural testing, functional testing is open-ended—the number of test cases that can be generated by the methods listed above is almost always very large, maybe even infinite. No analogues of coverage metrics exist to gauge the thoroughness of functional testing.

Automated Tool Support

Two types of automated tools support functional testing:

- Test harnesses (sometimes called test drivers)

- Stress testing tools

A test harness provides an environment for testing individual software modules as well as complete programs. The tool can fill in for missing program components, including a main program. Test harnesses are most useful in an interactive environment—there they can be used to start, terminate, or interrupt execution at an arbitrary point in a program. Most test harness tools have some debugging capabilities—these are discussed in Sec. 6.3. Test harness tools are listed along with other debugging tools in Table 6.5 (p. 6-13).

A stress testing tool such as GRC's Adaptive Tester[1] can automatically generate test data. The tool tries to find input data that will cause undesirable behavior in the test program. To do this, the user must come up with a numerical measure or program behavior—this is called an "objective function". Various techniques can be used to maximize (or minimize) the value of the objective function; most of these assume that the objective function has certain continuity properties.

5.4.2 Effectiveness

Functional testing can be used to detect all types of errors. The TRW analysis of combined functional and structural testing shows that a high percentage of most error categories can be detected. Special values testing alone detects a significant number of errors in many categories.

[1] C. G. Davis, "Testing Large, Real-Time Software Systems", Infotech State of the Art Report - Software Testing, Infotech International, Berkshire, England, Vol. 2, pp. 85-105.

The three error detection ratio data points for functional testing range from 50% to 90%. Functional testing can be used effectively at the unit testing stage. Automated stress testing has been shown to be a highly efficient way to detect errors.

Types of Errors Detected

In Sec. 5.3.2 we described the TRW study of the effectiveness of a combination of structural and functional testing for the Project 3 errors (the results are summarized in Table 5.11, page 5-46). Unfortunately, TRW did not distinguish between the two techniques. We cite the high percentage of errors detected in a wide range of error categories as evidence of the effectiveness of functional testing.

TRW did consider the use of special values testing alone on the Project 3 errors. This type of testing was also included in the study of the combination of structural and functional testing. Table 5.12 lists the error categories and percentages for special values testing.[1]

Error Detection Ratio Data

It is strange that relatively few studies of the effectiveness of the "traditional" testing method have been done, but that is the case. In addition to the result for combined structural and functional testing from the TRW Project 3 study (72.9% of total errors detected), we have only three other pieces of error detection ratio data to report for functional testing.

- Howden determined that functional testing could detect 14 of 28 errors (50%) in the sample from six programs in different languages.[2]

[1]Special values testing is called DSET (for Data Singularity and Extreme Test) testing in the TRW Report.

[2]W. E. Howden, "An Evaluation of the Effectiveness of Symbolic Testing," op. cit., p. 389.

TABLE 5.12

ERROR TYPES DETECTED BY SPECIAL VALUES TESTING

Source: Table 4-32 of the TRW Report

Error Category	Percent Detected
Computational	52.2%
Logic	57.3%
I/O	35.2%
Data Handling	84.2%
Routine/Routine Interface	20.0%
User Interface	87.3%
Global Variable/Compool Definition	81.3%
Documentation	23.5%

- Howden found that functional testing would catch 38 of 42 errors found in the IMSL programs, or 90%.[1] He noted that functional testing was the "best" technique to use for only 31 of those errors.

- The TRW Project 3 study determined that special values testing could detect 51.1% of all errors.[2]

Two other experiences with functional testing methods testify to its effectiveness. TRW used functional and structural testing at the unit testing level in Project 5, before the error data was accumulated.[3]

[1]W. E. Howden, "Functional Program Testing," op. cit.

[2]Table 4-33, p. 4-162 of the TRW Report.

[3]Pp 4-174 and 4-175 of the TRW Report.

They studied the error reports from integration testing to determine which errors should have been detected earlier in the life cycle than they actually were. They found that only 15.7% of these errors should definitely have been detected in the unit testing stage.

Benson and Andrews[1] performed an error seeding experiment in which forms of special values testing and stress testing were each combined with executable assertion testing. Only errors that could be detected by assertion testing with a nominal set of input data were considered. They compared the effectiveness and efficiency of the two types of functional testing. The input domain tests detected all of the errors considered; however, it required 683 test cases to do this. Stress testing, using the Adaptive Tester, detected all but one of the errors, and required only 57 tests; in fact, it detected all but two of the errors in just 7 tests!

5.4.3 Reliability

Since functional testing does not have a well-developed methodology or an objective measure of test thoroughness, its success depends heavily upon the skill of the person conducting the tests. Functional testing often operates under a budget constraint, in which case efficiency in finding errors is of utmost importance.

Any error that prevents a program from operating correctly can be found through functional testing. However, functional testing alone is not useful for determining the efficiency of a program or for debugging. Functional testing cannot guarantee the absence of errors or that the code has been thoroughly tested.

[1] J. P. Benson, D.M. Andrews, op. cit., pp. 1-11 to 1-12.

Functional testing works best when top-down project development is used. The greatest successes with functional testing have been for small, numerically-oriented programs. Many problems of testing large or non-numeric programs have not been addressed by current research.

Human Factors

Functional testing places a lot of analytical burdens on the tester. He must detect errors manually. He must select test data that will find errors effectively and efficiently. He must determine when the payoff of continued testing has become less than its cost.

The biggest accomplishment of recent research in functional testing is that new sources of test data have been identified. There is no cut-and-dried functional test methodology that can be followed; but if a tester thinks in terms of program functions, input domains, special values, and stress tests he should always be able to come up with a new test case to run.

We believe that if a tester understands the concept of functional testing, and understands the program being tested, he should be able to do effective testing. Understanding is the crucial word here--large programs require several people to contribute to the functional testing effort, since no one person can be familiar enough with the entire program to formulate test data and determine correct operation by himself.

Efficiency is a very important consideration in functional testing. Because of the large number of possible test cases, the amount of functional testing that is actually performed may well be determined by the amount of resources available for testing rather than by an objective measure of test thoroughness. In such a situation, there is a premium on choosing test data so that errors are detected as soon as possible. Advanced stress testing techniques, such as those used in the

Adaptive Tester, have shown major improvements in efficiency over manual methods. The success of such automated aids is still heavily dependent on the skill and judgment of the human tester.

Kinds of Errors

It is probably true that any error in a program can be found by functional testing. But it is also true that functional testing cannot guarantee that all errors in a program have been found. Unfortunately, beyond these two aphorisms little can be said about the reliability of functional testing for different error types.

Functional testing alone is not useful for determining the efficiency of a program. Dead code and extraneous uses of variables cannot be detected without knowledge of the structure of the program. The timing information provided by instrumentation tools is vital to improving the speed of execution of a program.

Even though functional testing is the most effective technique in terms of the number and types of errors detected, there are several reasons why it should not be the only test technique used. As we note in Sec. 6.1, functional testing does not give adequate information for locating and correcting errors. It is also important to use some structural coverage information during testing—otherwise, there is the danger that some sections of code will escape testing altogether.

Program Characteristics

Functional testing can begin at the unit testing stage of the life cycle if the units perform well-defined functions. For this reason, functional testing works well with top-down development. A test harness can be used to help build stubs for components that haven't been developed when testing begins.

Functional testing is difficult for programs that perform a large number of functions and have a lot of inputs. In such programs there are a large number of possible test cases. Programs in which the functions are highly interrelated can pose enormous difficulties: errors may mask each other, and the relationship between input and output can be very unclear. Good design techniques, especially modularization, can help overcome these problems.

Functional testing seems to work especially well for mathematical and numerically-oriented programs. The concepts of domain partitions, special values, and stress tests may not apply so easily to text processing, data base management, or other applications. Experiments with the functional techniques in these areas should be performed.

5.4.4 Cost

No one seems to be studying how much functional testing does cost or should cost. We found no data on either the amount of analysis time or the computer costs for specific applications of functional test methods. The relationship between the cost and effectiveness of increasing amounts of functional testing should be explored.

The cost of functional testing is most sensitive to the number of test runs made. This is true of both analyst and computer costs, since the set-up costs are low. Test harnesses and stress testing tools have very low overheads and can provide a net saving in computer and analysis costs over manual testing.

Analysis Time

Analysis time is likely to be the most significant cost item when doing functional testing. Most of the analysis time is spent examining test output for errors, although keeping track of past test results manually can become a burden. The amount of analysis time increases nonlinearly with the number of test runs made, since past test runs have to be considered when performing a new test.

Computer Resources

The computer resources needed to test a large program can be significant even if there is no tool overhead involved. This is because functional testing requires a large number of test runs for such programs. Test tools can help save computer resources by making it possible to test parts of code separately, by automatically keeping track of test results, and by helping to generate test data.

Test harness tools require a very small amount of overhead. During a test run, the tool supplies a skeleton substitute for missing modules, so execution time is much less than that required by the finished program.

Tools can digest test data and use it to guide the selection of new tests at negligible expense. We have used the Adaptive Tester as a test driver, stress testing tool, and data reduction package all at once when testing programs. We have used sophisticated test data generation algorithms and factor analysis on programs with up to 100 input variables. We have submitted jobs that made hundreds of runs of test programs on a CDC 7600. For test programs that required on the order of one second of CPU time per execution, we always found that the time required by the tool was a small fraction of the total execution time of the test run.

Procurement Costs

The XPEDITER tool package, which includes test harness and several other debugging capabilities, costs $25,000 to $35,000 depending on the options selected (permanent license fee).

Cost Worksheet

Functional testing is an open-ended testing technique--a reliability or cost criterion is needed to tell when to stop testing. The amount of testing that can be done depends on the cost per test run. There is a heavy premium on efficiency in choosing test data that detects errors early in the testing process.

5.5 FORMAL TECHNIQUES

5.5.1 Summary

The formal techniques are symbolic execution and formal verification (program proving). Symbolic execution is the process of forming mathematical expressions that relate the inputs and outputs of a program. These expressions can be used to determine whether a program correctly implements an algorithm; they can also be useful in static and structural testing.

Formal verification means proving that a program complies with its specifications. Formal verification dominates the development of any software project in which it is used, since it affects the requirements, design, and coding of the program. Formal verification must be planned *for from the beginning of a project*--it cannot be added as an afterthought.

Tools have been developed to support the formal techniques. Sophisticated mathematical and artificial intelligence techniques are used in these tools. Still, they must be guided closely by the user. Existing tools are not suitable for use outside of a research laboratory.

Symbolic Execution

Symbolic execution attempts to derive mathematical expressions for the outputs of a program in terms of the input variables. Conceptually, this can be done by carrying out the actions specified by each executable statement in a program while storing symbolic instead of actual values for variables. For example, a loop to sum the elements of an array

```
SUM = 0.0
DO (I = 1,4)
       SUM = SUM + A(I)
END DO
```

when symbolically executed would yield the expression:

```
SUM = A(1) + A(2) + A(3) + A(4) .
```

Symbolic execution has a number of applications in program verification:

- Programs can be tested symbolically by comparing the expressions derived for output variables with the desired formulas.

- The conditions that cause a particular program path to be executed can be determined symbolically. These path conditions can then be used to generate actual test data values for use in structural testing.

- The path conditions can also be used to determine which paths are feasible—that is, which ones can actually be executed. This information is useful to both static analysis and structural testing.

- Symbolic execution is used in the course of doing formal verification of programs.

Formal Verification

Formal verification means constructing a mathematically rigorous proof that a program will behave according to its specifications. A complete proof of a program examines both the code and specifications and shows that they are consistent. Walker, et al.[1] describe one way to achieve a complete formal verification of some program properties. Their method consists of three separate steps:

- A top-level specification for the program is defined. This is a concise statement about what the program is supposed to do. An abstract-level specification that contains a general description of the design of the program is then formulated. The consistency of the abstract-level and top-level specifications is proved.

- A low-level specification is developed which contains enough information to enable the code to be verified. The consistency of the low-level and abstract-level specifications is proved.

- Finally, a code-level proof is performed. This is usually done by the standard Floyd-Hoare methods.[2] The low-level specification provides the input and output assertions that are used to form the program's main verification conditions.

Formal verification is different from testing in that both the specifications and the code are critically examined. Changes need to be made in the specifications and design as well as the code in the course of doing verification. Because of this, the entire program development effort must be dedicated to the task of proving the program. Programs that are not designed and written to be formally verified cannot be proved.

[1] B. J. Walker, R. A. Kemmerer, G. J. Popek, "Specification and Verification of the UCLA Unix Security Kernel", Communications of the ACM, Vol. 23, No. 2 (Feb. 1980), pp. 118-131.

[2] A good explanation of Floyd-Hoare techniques is contained in S. L. Hantler and J. C. King, "An Introduction to Proving the Correctness of Programs," Computing Surveys, Vol. 8, No. 3 (Sept. 1976), pp. 331-353.

During the course of proving a program, errors in code or specifications may be detected. Program proofs are conducted in a series of sub-proofs or lemmas; each lemma builds on the previous ones by examining a relatively small number of additional statements. When one of the lemmas can't be proved, the reasons for the failure may provide information about what changes can be made to correct the situation and allow the proof to succeed.

Prof. Richard Kemmerer related[1] an example of a graduate student who was helping with the verification of a communication system. The student knew very little about how the system worked, but found a specification error in the course of trying to complete a proof.

Automated Tool Support

Four kinds of tools support the formal techniques:

- Symbolic execution tools
- Verification condition generators
- Theorem provers
- Proof verifiers

A symbolic execution tool has two major components: a program interpreter and an expression simplifier. The interpreter translates each successive action of a program into a mathematical algebraic expression. The simplifier then tries to reduce the output expressions to as simple a form as possible. Symbolic execution is usually done interactively, so that the user can guide the tool in each of these tasks.

Verification condition generators use symbolic execution techniques. They work backwards from the final assertion of a lemma and derive the necessary preconditions for it to hold.

[1]Personal communication, June 1981.

Once the verification conditions are formed, a theorem prover can be invoked to try to establish the truth of the lemma. Theorem provers are complex programs which can interpret the assertions and program statements and apply complex rules of inference and various heuristic proof strategies.

Once a proof of a lemma has been constructed, it must be checked for correctness. Theorem provers usually have a proof verifier built into them. Special verifier tools exist to check manually generated proofs.

Formal techniques and the tools that support them are still being developed by researchers. Formal techniques are so complicated and specialized that no tools have yet been developed that are suitable for wide-spread use. Institutions that are conducting research on formal techniques usually develop their own "proof systems," which typically include special programming languages and design methods as well as the tools mentioned here. Several such systems are described in the tools survey in Appendix B.

5.5.2 Effectiveness

Symbolic execution can be used to test a program in a manner similar to structural and functional testing. A significant number of errors in a program can be detected this way.

In the last two years, several successful applications of formal verification to large, useful programs have been announced. These have been proofs of the security of computer or communication systems. A major shortcoming of current formal verification methods is that there is no known way to prove fault-tolerance.

Symbolic Execution

Howden has studied the use of symbolic execution to enhance other test techniques.[1] He considered the effect of using functional and structural testing methods on symbolic as well as real data. Symbolic structural testing means deriving algebraic expressions for those paths that were selected during structural testing with actual values. Symbolic functional testing means deriving algebraic expressions for the program functions.

Howden examined the errors from his sample of six programs in four different languages. He found that symbolic testing alone could detect 17 of the 28 errors, or 61%. Symbolic testing improved the effectiveness of both structural and functional testing

Successful Formal Verification Efforts

In the past few years researchers in the field of formal verification have felt challenged to prove the usefulness of their methods in the "real" world. Recently they have enjoyed quite a few successes for programs of respectable size and complexity.

Walker, et al. completed most of the verification of the security of the UCLA Unix Kernel. The kernel forms part of a working operating system for a PDP-11/45, although the system is unacceptably slow. (The reasons for the poor performance are unrelated to the fact that it was subjected to verification.) The project was intended to be strictly for research and demonstration, so not all of the proofs were carried out. The participants are convinced that the rest of the proof could be completed, given sufficient resources.

The notes of the first two Verification Workshops contain descriptions of two other successful recent verification efforts. These were:

[1] W. E. Howden, "An Evaluation of the Effectiveness of Symbolic Testing", op. cit., p. 389.

- Verification of a security guard in a distributed system. The system is working with the guard embedded in it.[1] This work was done at the Institute for Computing Science and Computer Applications of the University of Texas.

- A year later, the same researchers reported that they had verified an encrypted packet interface for the ARPANET.[2]

What Properties Can Be Verified?

Recent verification efforts have concentrated almost exclusively on the security properties of computer and communication systems. The methods of verifying such systems have become well known in the formal verification community. However, there are other software application areas in which performance is so critical that the expense of verification would be justified if there were a reasonable chance of success.

Boebert[3] challenged the gathering at the VERkshop to consider what would be involved in verifying embedded control software such as in an aircraft autopilot system. He notes that such systems are computationally simple, use simple data types, and maintain a small state space. However, the verification requirements are severe and there is the need to prove fault-tolerance.

Haynes[4] also discussed potential applications of formal verification. He noted that two shortcomings of current verification methods were an inability to analyze the accuracy of floating point computations and the lack of a way to specify fault-tolerance requirements.

[1] D. I. Good, M. K. Smith, "A Verified Distributed System", mimeographed notes of the Verification Workshop ("VERkshop"), SRI International, Menlo Park, CA, April 1980.

[2] M. K. Smith, et al., "A Verified Encrypted Packet Interface", Software Engineering Notes, Vol. 6, No.3 (July 1981), pp. 14-16.

[3] W. E. Boebert, "Formal Verification of Embedded Software", first VERkshop notes, p. 106.

[4] G. A. Haynes, "Position Paper on Program Verification", first VERkshop notes, p. 9.

5.5.3 Reliability

The shortcoming of formal techniques is that they are so complicated that user errors are inevitable. Currently tools do not remove enough of the mechanical burden from the user. Both symbolic execution and program proving require a lot of tedious work. The expressions generated by formal tools are often complicated and difficult to interpret.

Howden drew the following conclusion from his experience with symbolic testing:

> "In general, it was found to be difficult to apply symbolic evaluation to all but the functional modules at the lowest level of abstraction...symbolic evaluation should be limited to low level modules..."[1]

It is not surprising that mistakes have been made in the use of program proving techniques. Gerhart and Yelowitz[2] discuss four programs that were claimed to be proved and were later found to contain errors.

A major problem with formal verification is that a set of formal specifications must be written for a program. This set of specifications must be complete and rigorously formulated, since the proof will be based upon it. Writing formal specifications is a difficult and tedious task--it is difficult to find people capable of and willing to do this work.

[1] W. E. Howden, "An Evaluation of the Effectiveness of Symbolic Testing," op. cit., p. 395.

[2] S. L. Gerhart and L. Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies," IEEE Transactions on Software Engineering, Vol. SE-2, No. 3 (Sept. 1976), pp. 202-205.

Formal verification imposes some constraints on the form of the software. Some researchers feel that a code-level proof is only valid if the code is written in a language that is axiomatically defined. This excludes more commonly-used languages such as FORTRAN and COBOL, although other researchers have used formal techniques with these languages. In all programming languages, there are serious problems with proving properties of programs that include real or character variables or that have timing or synchronization logic.

5.5.4 Cost

The formal techniques are very expensive compared to static and dynamic testing because they involve intensive amounts of highly skilled labor. Because the formal techniques are still being developed, there are no good ways of estimating the costs of a prospective application. Project managers who are considering using formal techniques should obtain the services of someone who is experienced in the field.

Symbolic Execution

Howden[1] gives some estimates of the time and storage required by symbolic execution tools. To symbolically execute a path through a program, the storage required is proportional to the sum of the number of branches in the path, the number of statements in the path, and the number of variables in the path.

Formal Verification

Below we present some information on the amount of effort required by a few program proving efforts. A major problem with this data is that it is difficult to make generalizations across projects. Different people approach formal verification in different ways. Complete verifications are not always performed--some just do a code-level proof;

[1]W. E. Howden, Symbolic Testing--Design Techniques, Costs and Effectiveness, NTIS PB-268, 517, U.S. Department of Commerce, Springfield, Virginia.

others do not rigorously apply the proof methodology to the entire program and specifications.

Another problem with determining "representative" costs of formal verification is that there is no agreed-upon standard of the size of the job. Measures such as the number of lines of code or the number of modules do not give a good indication of the difficulty of proving a program. Some of the data cited below give the number of verification conditions used in a proof. Since different proof techniques yield different numbers of verification conditions, this does not provide a universal standard, either.

- Walker, et al.[1] required four to five person-years on the UCLA Secure UNIX kernel.

- Smith, et al.[2] report that they used only two person-months in completing their verification of the Encrypted Packet Interface. The proof required 185 verification conditions, of which 144 were proved automatically by the algebraic simplifier tool. The program contains about 2000 lines of Gypsy code.

- A small queue manager program was verified using the SQLAB tool at GRC[3]. Six verification conditions were generated--two were proved automatically, the others interactively. This work required 21 seconds of execution time on a CDC 7600.

[1] B. J. Walker, et al., op. cit., p. 128.

[2] M. K. Smith, et al., op. cit.

[3] S. H. Saib, et al., op. cit., p. 179.

- As an indication of how the performance of verification tools can be improved, consider the experience of SRI with the RPE (Rugged Programming Environment) proof systems[1]. The two versions were benchmarked using a binary search routine. The earlier version (RPE/1) required 210 seconds for parsing, 5 seconds for verification condition generation, and 600 seconds for proof deduction on a DEC KA-10 computer. The improved version (RPE/2) required 0.7 seconds for parsing, one second for verification condition generation, and 27.5 seconds for proof generation, although these figures are for a slightly faster machine. The overall improvement in speed is a factor of about 27 to 1.

[1]B. Elspas, R. E. Shostak, J. M. Spitzen, A Verification System for JOCIT/J3 Programs (Rugged Programming Environment - RPE/2), Rome Air Development Center Technical Report No. RADC-TR-77-229, p. 35.

6 OTHER CAPABILITIES OF TEST TECHNIQUES AND TOOLS

We selected our breakdown of test technique categories to facilitate the analysis of error detection. Some of these techniques also provide information that is helpful in the tasks of error location and correction. We describe this information in the first two subsections below. In Sec. 6.3 we look at some automated tools that have been developed specifically as aids for error location and correction. Many of these tools also support one or more of the test techniques.

Of the static and dynamic test techniques, static analysis clearly provides the greatest amount of debugging information about the errors that it detects. Functional and structural testing provide the least. In between these extremes lies executable assertion testing, whose debugging capabilities have never been studied in detail. Although we know of no empirical evidence about the debugging effectiveness of the test techniques, we offer the ranking of the techniques for error location and correction capability shown in Table 6.1.

TABLE 6.1
ERROR LOCATION AND CORRECTION CAPABILITY RANKINGS
(1 = best)

1. Static analysis
2. Executable assertions
3. Structural testing
4. Functional testing

We have not included formal verification in our treatment of debugging. However, program proving can produce error correction information in the same way that it can detect errors. We described how errors are detected during verification in Sec. 5.5.1.

The techniques described in this report can be useful in two other areas of software development besides testing: program documentation and the analysis of program quality. In Sec. 6.4 we look at how some of the documentation required by a military standard can be satisfied by static analysis reports. In Sec. 6.5, we examine the ways that the test techniques support two schemes for evaluating the overall quality of a piece of software.

The test techniques supply less than half of the information required by either the documentation standard or the quality evaluation schemes. However, static analysis provides significant amounts of information in both areas. Most of the information required for documentation and program analysis can be obtained from source code. Therefore, static analysis tools could be expended to provide this information.

6.1 ERROR LOCATION

Both static analysis and executable assertion tools give statement numbers in their diagnostic messages. However, assertion diagnostics represent symptoms of errors, while static analysis can often identify the source or instance of an error. The kinds of error and anomaly checking that static tools can perform are discussed in the characteristic profile in Sec. 5.1. Static analysis cannot always isolate errors to single statements, but it usually does so for the following error and anomaly types:

- Module interface errors
- Coding standards violations
- Mixed-mode arithmetic

Other static error and anomaly types by their nature span a range of statements. However, a few of the statements in the range can be singled out for closer scrutiny. For example, if a variable is reported to be referenced at a statement before it has been defined, it may be that an assignment or other defining statement needs to appear somewhere on the path to the reference. But it is also possible that the statement referencing the variable has been coded incorrectly, and another variable should appear in place of the undefined one. Thus, static analysis provides localization of errors to the statement, in the best case—<u>or</u> to the path, in the worst case—for the following error and anomaly types:

- Uninitialized variables
- Variables set but not used
- Unreferenced statement labels

A third class of errors and anomalies detected by static testing reflect problems in program logic or control flow. Correcting these types of problems may just require a change in one statement; or it may be necessary to rewrite a fairly large amount of code. These errors and anomalies include:

- Infinite loops
- Recursive procedure calls
- Deadlock
- Unreachable code

For static tools to detect these errors, they must be structural —that is, the errors must be inherent in the control structure of the program, and not depend upon relationships between variables. Usually the changes required to correct such problems will be made to code that is within or near the range of an error. This is because structural logic errors are normally the symptoms of simple acts of negligence on the part of the programmer. These error types are much harder to locate if they cannot be detected structurally.

6.1.1 Error Location Metric

The preceding analysis suggests a metric for the degree of localization of errors that the static techniques provide. For each of the three groups of static error and anomaly types listed, we can examine the ratio:

$$\frac{\text{Amount of code that must be searched to find the error}}{\text{Amount of code which is incorrect}}$$

An error in the first group of static checks listed above is usually located at the statement for which a diagnostic message is issued. Thus the error location metric value for these error types will usually be one.

An error from the second group may also be located at a statement given in the diagnostic; or it may be the case that the code that (logically) precedes the statement must be searched to find the error. Thus the size of the numerator of the location metric for this group can vary from as little as one statement to as much as an entire module, or several modules for global variables. Since the code change required is usually on the order of one statement, the location metric value will vary considerably for these error types.

The last group of errors usually involves a code segment such as a loop or branch.[1] The statements that cause the error often are control points (e.g. "if", "while", "goto" statements) in the vicinity of the

[1]We use the term "branch" to mean all statements between two logically adjacent control points in a program.

identified segment. So the location metric value for these errors should typically be small.

Table 6.2 summarizes the error location metric as applied to the static error and anomaly types. Structural descriptions of code (statement, branch, path, module) are used instead of numbers since they are more descriptive in this situation.

6.1.2 Error Location Capabilities of the Dynamic Techniques

As mentioned above, executable assertions also supply statement numbers with their diagnostic messages. This feature can be used by a skillful tester to provide a great deal of information about the location of certain kinds of errors. For example, when assertions are used to check the validity of steps in a complicated computation, a report of an assertion violation often isolates the error to the last step performed before the assertion.

However, since assertions can only impose conditions involving values of program variables, they report symptoms of errors rather than causes. There may be a lot of code separating the point where a bad value is generated and the place where it is used in a computation that is checked by an assertion. It may be that the value did not look bad when it was generated, but caused a problem because it was used incorrectly. For example, assertions can be used to check for division by zero. When a programmer finds that his program is guilty of producing a zero as a denominator, he does not always change code that is close to the statement that performs the division.

TABLE 6.2

LOCATION METRIC APPLIED TO STATIC ERROR AND ANOMALY TYPES

Error or Anomaly Type	Location Metric Value
Module calling sequence error	One or a few statements One statement
Coding standards violation	
Mixed-mode arithmetic	
Uninitialized variable	One statement to several modules One statement
Variable set and not used	
Unreferenced statement labels	
Infinite loop	One statement to a few branches One statement to a few branches
Recursive procedure call	
Deadlock	
Unreachable code	

As was the case for error detection, the effectiveness of assertions for locating errors depends to a great extent upon the skill and knowledge of the person who writes the assertions. We feel that assertions which provide effective error detection will also supply good error location information. However, to best locate errors, individual assertions should contain as few variables and relations as possible. For example, the statements

```
ASSERT (M.GT.0)
ASSERT (N.LE.100)
```

provide better location information than the single assertion

```
ASSERT ((M.GT.0).AND.(N.LE.100)),
```

even though both sets will detect the same errors.

We believe that an effective set of assertions will isolate most detected errors to the segment of code between the last branch point and the position of the violated assertion. In terms of the error location metric, this means assertions localize errors to the level:

$$\frac{\text{Number of statements on a branch}}{\text{One statement}}$$

We base this conjecture on a standard for using assertions that requires all program inputs and each control point to be checked for validity of relevant variable values. This standard is part of the assertion methodology proposed in Sec. 7.2. If our conjecture holds, then assertions are a fairly powerful error locating technique.

Structural testing can provide a much smaller amount of error location information. No diagnostic messages are associated with this technique; however, the coverage reports issued by instrumentation tools indicate what sections of code were executed in a test run. Any code that is in error is obviously part of the code that is executed. Each test case represents one complete path through the program, which may be a very large amount of code.

Functional testing alone provides no error location information. If this technique is being used to test a small amount of code--for instance, a single module during unit testing--then this may not be a serious drawback to its use. Functional testing may sometimes be used alone when debugging does not accompany testing--for example, during acceptance testing, in which errors are noted on problem reports for later consideration. But to test and debug medium-sized or large programs, functional testing should be augmented either by another test technique which provides error location information, or by specialized debugging tools as described in Sec. 6.3.

6.2 ERROR CORRECTION

Automated tools cannot correct errors in the sense of making changes to code, but they can help a programmer make corrections. Since the diagnostics issued by static analysis describe wrong or questionable conditions in a program, they suggest actions which might remedy the conditions. The test techniques can also be used to help to verify that a change made by the programmer has corrected an error and has not introduced other errors.

Table 6.3 lists some of the corrective actions suggested by each of the static error and anomaly checks. These actions will not always solve the problem (indeed with an anomaly there may be no problem at all) but they may provide a starting point in the search for a correction. Other corrections may be suggested by combinations of static messages: for instance, a misspelled variable will usually produce both "uninitialized variable" and "variable set but not used" messages, one each for the correctly spelled and misspelled versions.

TABLE 6.3

CORRECTIVE ACTIONS SUGGESTED BY STATIC ANALYSIS DIAGNOSTICS

Error or Anomaly Type	Possible Corrective Action
Infinite loop	Add a statement checking for exit condition
Module calling sequence error	Change the variables in the "CALL" statement
Recursive procedure call	Convert code to nonrecursive form *
Uninitialized variable	Add an initializing statement
Deadlock	Provide for supervision of "wait" state
Coding standards violation	Use equivalent standard construct
Mixed-mode arithmetic	Declare variables of proper type
Variable set and not used	Remove the last assignment
Unreachable code	Remove the code segment
Unreferenced statement label	Remove the label

* A standard method of removing recursion is given in E. Horowitz and S. Sahni, Fundamentals of Data Structures, Computer Science Press, Inc., 1976, pp. 160-161.

An Error Correction Metric

When a programmer makes a change to his code to correct an error he will normally rerun some tests to see that the change has corrected the problem. If this retesting is to be effective, all the tests that may be affected by the code change must be rerun. Although this will not guarantee that the changed code is correct, it does update the entire testing process to the point at which the last error was found.

For a technique to be a good debugging aid, it must be easy and inexpensive to do retesting. The effort required to retest a program depends on the nature of the test technique and the code changes that were made. This can be stated in the form of a metric as:

$$\frac{\text{(Number of retesting runs) x (Amount of code in a retest run)}}{\text{Amount of code changed}}$$

For most kinds of static testing, only those modules in which a change appears must be rerun through the static test tool. This is true even for changes that affect global variables, because static tools can supply the information about the behavior of other modules from a stored data base.[1] So the retesting metric value for static analysis is one test of one module per module changed.

To effectively retest using dynamic test techniques, all the test runs affected by the changed statement should be rerun. For structural testing, there is information available about which previous test cases executed the changed statement. Only these cases need be rerun. This information is not available when functional or executable assertion testing are used alone, so all previous test cases must be rerun.

If information were available about when in the program execution sequence the changed statement appeared, and if the program could be restarted at that point, then each dynamic test run would not have to be repeated in its entirety. Test harness and interactive debugging tools have some of these capabilities (see Sec. 6.3), but we know of no tool which is entirely suitable for this purpose.

[1]The data base library feature of the SQLAB tool is described in S. H. Saib, et al., Advanced Software Quality Assurance Final Report, General Research Corporation CR-3-770, May 1978, pp. 143-144.

Table 6.4 summarizes the error correction metric for the static and dynamic test techniques. For the dynamic techniques, the amount of retesting depends upon the amount of testing performed previously. This puts a premium on finding and correcting errors early.

TABLE 6.4

CORRECTION METRIC APPLIED TO THE STATIC AND DYNAMIC TECHNIQUES

Test Technique	Correction Metric Value
Static analysis	One test of one module per module changed
Structural testing	All affected test cases
Executable assertions	All test cases
Functional testing	All test cases

6.3 DEBUGGING TOOLS

A considerable number of debugging tools and packages have appeared in recent years. Many of the tools are available commercially and are described in software product directories such as Datapro.[1] These tools usually provide some combination of five types of features:

- Formatted dumps
- Execution traces
- Test harness capabilities

[1] Datapro Directory of Software, Datapro Research Corporation, August 1980.

- Interactive debugging
- Test languages and libraries

Formatted dumps augment the traditional method of debugging programs by examining octal or hexadecimal dumps of memory areas at program termination. A tool can attach variable names to values and properly interpret numeric and character data. It can also give the immediate cause of an abort, the source code module name and line number, and the sequence of module invocations at the time of the termination.

Execution traces provide a log of program operation during the entire course of execution. Statements covered, modules called, and changes in the values of specified variables are reported. Execution traces can give information about nearly every facet of program operation, and thus are potentially a powerful test technique. However, they tend to overwhelm the user with output and thus do not reduce the analytical burden of testing.

Test harnesses allow the tester to exercise parts of programs, by providing stubs or other substitutes for missing modules. We consider test harnesses to be the main automated support for functional testing. They can also be used to isolate crucial or troublesome areas of code for more thorough testing.

Interactive debugging gives the tester enhanced control over program execution. He can specify places in the program where execution is to stop so that conditions can be examined. The tool saves the program state, so that testing can be restarted at that point. A tool can also permit execution to be started at an arbitrary entry point by querying the user for needed information.

Test languages and libraries provide a way for a set of tests to be automatically documented and reproducible. This can automate the retesting process and insure a great degree of thoroughness. Tools can also generate graphic displays of test results that are useful for management purposes.

These five debugging features can be combined with the test techniques described in this report to form more powerful test tools. A drawback to incorporating the debugging features into general-purpose tools is that often their implementation is highly system-dependent. Table 6.5 lists some tools with these debugging features that were included in the tools survey in the first interim report. Many other tools are available commercially, most of which are designed for widely used business computing systems.

TABLE 6.5
TEST TOOLS WITH DEBUGGING CAPABILITIES
(Source: Tools Survey, Appendix B)

Tool	Capabilities
ATDG	Test harness
CAVS	Execution tracing, test library
OPTIMIZER III	Dump formatting, interactive debugging
PRUFSTAND	Dump formatting, execution tracing, test harness, interactive debuggging
TESTMANAGER	Dump formatting, test harness
TPL	Test harness, test language and library
XPEDITER	Formatted dumps, execution tracing, test harness, interactive debugging, test language and library

6.4 DOCUMENTATION

A by-product of many static analysis reports produced by automated testing tools is information which can be used to help document the program being analyzed. A summary of the kinds of information produced was given in the first interim report and is reproduced below.

- Global cross-reference report indicating input/output usage for variables in all modules
- Module invocations report indicating the calling modules and showing all calling statements
- Module interconnection report showing the program's module calling structure
- Special global data reports for variables in COMMON blocks and COMPOOLs
- Program statistics including total size, number of modules, module size distribution, statement type distribution, and complexity measures
- Summaries of analysis performed, program statistics, and errors and warnings reported

As an example of a standard for adequate documentation of computer programs, we will use MIL-STD-483, Appendix VI.[1] The relevant sections of this appendix are those describing the Computer Program Configuration Item (CPCI) Part II specification. An outline of the required content of this specification is shown in Table 6.6. Sections 3 and 4 are relevant to documentation of computer programs.

[1] Configuration Management Practices for Systems, Equipment, Munitions and Computer Programs, MIL-STD-483 (USAF) Notice 2, March 21, 1979.

TABLE 6.6

MIL-STD-483 CPCI PART II SPECIFICATION

1. Scope
2. Applicable Documents
3. Requirements
   - 3.1 CPCI Structural Description
   - 3.2 Functional Flow Diagrams and Charts
   - 3.3 Interfaces
   - 3.4 Program Interrupts
   - 3.5 Timing and Sequencing Description
   - 3.6 Special Control Features
   - 3.7 Storage Allocation
     - 3.7.1 Data Base Definition
       - 3.7.1.1 File Description
       - 3.7.1.2 Table Description
       - 3.7.1.3 Item Description
       - 3.7.1.4 Graphic Table Description
       - 3.7.1.5 CPCI Constants
     - 3.7.2 CPC (Computer Program Component) Relationship
     - 3.7.3 Data Base Location Requirements
   - 3.8 Object Code Creation
   - 3.9 Adaptation Data
   - 3.10 Detail Design Description
     - 3.10.X Identification of CPC No. X
       - 3.10.X.1 CPC No. X Description
       - 3.10.X.2 CPC No. X Charts
       - 3.10.X.3 CPC No. X Interfaces
       - 3.10.X.4 Data Organization
       - 3.10.X.5 CPC No. X Limitations
       - 3.10.X.6 CPC No. X Listings
   - 3.11 Program Listings Comments
4. Quality Assurance
   - 4.1 Test Plan/Procedure Cross Reference Index
   - 4.2 Other Quality Assurance Provisions
5. Preparation for Delivery
   - 5.1 Preservation and Packaging
   - 5.2 Markings
6. Notes

10. Appendix(es)

Table 6.7 shows for each static analysis report, the section of the requirements for which it produces documentation.

Executable assertion testing is the other technique that is a source of program documentation. Assertions provide in-line documentation of a program, in a manner similar to comment statements. Assertions are not a substitute for comments as explanatory material. However, they do provide very helpful information.

TABLE 6.7
DOCUMENTATION PRODUCED BY STATIC ANALYSIS TECHNIQUES

Static Analysis Report	Part II Specification Section(s)
Global cross-reference report	3.7.2; 3.10.X.3; 3.10.X.4
Module invocations report	3.3; 3.10.X.3
Module interconnection report	3.1
Special global data report	3.10.X.3
Program statistics	none
Summaries of analysis performed	4.1; 4.2

6.5 CONTRIBUTION OF TEST TECHNIQUES TO SOFTWARE QUALITY EVALUATION

A high quality software product does more than just satisfy its specifications. It should be efficient; it should be easy to understand, use, and maintain; and it should be flexible enough to be used in different environments. Two schemes which have been developed to evaluate these program qualities are briefly described here. We then examine how automated test techniques can be used to measure these qualities.

TRW Study

The first evaluation scheme was developed at TRW.[1] A set of seven desirable qualities for programs was identified. These desirable qualities are evaluated in terms of twelve "primitive characteristics" that are measured by examining the program. Each primitive characteristic is associated with a list of questions about the program--the characteristic is measured by answering the questions.

Test techniques can support the evaluation of the primitive characteristics in three different ways. Some of the questions are normally answered during testing--we have termed this "direct" support. For example, one of the "completeness" questions is: "Is the code free of obvious errors?" This question can be answered by applying static and dynamic testing to the program. If a program "passes" a thorough test sequence, then it will have all of those properties which are directly supported by the test techniques used.

In other cases, test techniques can provide the information needed to answer a question, but this information is not necessarily used as part of the testing process. We have termed this "indirect" support. For example, one of the "structuredness" questions is: "Are the modules limited in size?" Static analysis tools, or compilers that provide program listings, can give the length of each module in a program. However, standards of program size may not automatically be used in static testing,[2] so static testing does not guarantee that a program has this structuredness property.

1 B. W. Boehm, et al., Characteristics of Software Quality, TRW Systems Group Report No. TRW-SS-73-09, December 28, 1973.

2 Many standards checkers call attention to a module that has an excessive number of executable statements by printing an error message. This would constitute direct support for this structuredness question. However, most general-purpose static tools do not enforce module size standards: they typically only give size statistics as part of documentation reports.

A third way for test techniques to support quality characteristics is by providing a required capability to a program. Consider the completeness question: "Is input data checked for range errors?" If executable assertions which check variable ranges have been added to the code, then this question can be answered "yes". We call this "supplementary" support for a characteristic.

Using our knowledge of test technique and tool capabilities, we have rated the degree of support provided to each of the TRW primitive characteristics. The results are presented in Table 6.8. Support for a characteristic is rated high (H) if most of the questions for that characteristic can be answered from information provided by the test techniques. Support is rated medium (M) if there is considerable support but there remains important information that cannot be supplied by the techniques. A characteristic receives at least low (L) support if test techniques can answer any of its questions - otherwise it receives a zero (0) for no support.

The way in which support is provided is listed as direct (D), indirect (I), or supplementary (S). Since each primitive characteristic may have several questions associated with it, more than one type of support may be listed.

The most common supporting test technique is static analysis. This reflects the fact that many of the TRW questions are based on properties of the source code. The dynamic techniques of functional testing, executable assertions, and instrumentation play an important part in supporting four of the characteristics. Very few questions were found which could be answered by more than one technique.

TRW considered two methods of collecting data about the quality factors: an algorithm which scans source code without user input or guidance, and a consistency checker which requires the user to supply a

TABLE 6.8
TEST TECHNIQUE SUPPORT FOR TRW QUALITY FACTORS

Quality Characteristic	Fraction Automatable (TRW)	Degree of Support	Method of Support	Test Techniques
DEVICE-INDEPENDENCE	4/6	M	I	Static statement cross-reference Standards checker-language, portability
COMPLETENESS	12/15	M	S,D,I	Static data flow analysis Standards checker-language Executable assertions Functional testing
ACCURACY	0/1	H	D	Functional testing
CONSISTENCY	7/12	M	D	Static mixed mode analysis, global data documentation Standards checker - language
DEVICE-EFFICIENCY	3/4	M	I	Instrumentation - timing analysis
ACCESSIBILITY	[illegible]/3	0	-	—
COMMUNICATIVENESS	9/11	M	S	Test harness Executable assertions Instrumentation - execution tracing
STRUCTUREDNESS	7/9	M	D,I	Static module interface analysis, calling tree documentation Standards checker-language
SELF-DESCRIPTIVENESS	3/11	L	S,I	Static units assertions Functional testing
CONCISENESS	7/8	M	D	Static unreachable code detection, statement cross-reference
LEGIBILITY	9/10	M	D,I	Static statement cross-reference Standards checker-language
AUGMENTABILITY	1/1	0	—	—

checklist. Such tools need not currently exist for a question to receive a favorable rating on automatability—TRW only required that it be cost-effective to build and use them. They determined that 72% of all the questions, and also 72% of the important questions, could be answered partially or completely by such tools. We judged that 42% of all questions, and 47% of the important questions, were supported by test techniques.

GE Study

The GE software quality evaluation scheme[1] has a structure similar to the TRW scheme. A somewhat different set of quality characteristics, called "software quality factors", were selected. A set of software metrics was developed to support the evaluation of the quality factors. The major differences between the GE scheme and TRW's are that several quality factors may share the same metrics, and the metrics may take numerical values rather than just yes or no.

Our evaluation of the support that test techniques can provide for the GE quality factors is presented in Table 6.9. As in the TRW study, various static analysis reports provided the greatest amount of support. The dynamic analysis techniques play a much smaller role in supporting the GE quality factors than they did in the TRW scheme.

It is surprising that correctness is given a low degree of support by the test techniques. The GE metrics for correctness are: traceability of modules to requirements, completeness, procedure consistency, and data consistency. It is difficult to understand why no operational standards were included under correctness.

[1] J.A. McCall, P.K. Richards, and G. Walters, Factors in Software Quality, Metric Data Collection and Validation, General Electric Co. Report No. RADC-TR-77-369, Vol. II, November 1977.

TABLE 6.9
TEST TECHNIQUE SUPPORT FOR GE QUALITY FACTORS

Quality Characteristic	Fraction Automatable (GE)	Degree of Suport	Method of Support	Test Techniques
CORRECTNESS	10/15	L	D	Static data flow analysis, module interface analysis
RELIABILITY	14/47	M	I,S,D	Static data flow analysis, statement cross-reference, flow charts Executable assertions
EFFICIENCY	14/20	L	D	Static data flow analysis, anomaly detection, units assertions
USABILITY	0/23	0	—	—
INTEGRITY	0/5	0	—	—
MAINTAINABILITY	28/46	M	I,D	Static data flow analysis, statement cross-reference, flowcharts, calling tree Standards checker-language
FLEXIBILITY	20/29	L	I	Static statement cross-reference, calling tree Standards checker-language
TESTABILITY	28/44	M	I,D	Static statement cross-reference, calling tree, data flow analysis Structural testing Instrumentation-timing analysis
REUSABILITY	21/30	M	I,D	Static statement cross-reference, calling tree Standards checker-language
PORTABILITY	16/25	M	I,D	Static statement cross-reference, calling tree Standards checker-language
INTEROPERABILITY	16/26	M	I,D	Static statement cross-reference, calling tree Standards checker-language

GE judged that between 60% and 70% of the metrics were automatable for eight of the quality factors. Overall, 40% of the metrics were rated automatable. We found that 32% of all metrics were supported by test techniques. None of the quality factors had more than 48% of their metrics supported by test techniques.

The amount of test technique support for the TRW and GE software quality categories is rather disappointing. However, the two studies emphasized structural rather than operational properties of software. A thorough test exercise of a program will tell more about how correct and reliable a program is than the "medium" ratings indicate. Someone who has spent a lot of time performing tests will have a good idea how easy a program is to use, test, and modify. However, the low ratings for other quality factors mean that testing will not guarantee that a program has these desirable qualities.

Both the TRW and GE studies show a greater potential for automated data collection than what is currently provided by the test techniques. These data collection capabilities could be incorporated into the standard test tools to insure greater overall quality in tested software.

7 PROSPECTS FOR IMPROVEMENTS IN THE TEST TECHNIQUES

The ratings in the previous sections reflect the current state of refinement of the test techniques. The ratings are based on the way the tools are used individually in a typical software development environment. This section describes some ways that the techniques can be improved. In some cases, these improvements are still being developed and refined by researchers. But in other cases the methods are available now.

Advanced software engineering methods can ameliorate some problems associated with testing. For example, static testing of a program written in the language Ada will produce no extraneous mixed-mode warnings (indeed no mixed-mode warnings at all)--Ada requires strict type compatibility in all expressions and assignment statements.

Combining several test techniques can relieve some of the analytical burdens of testing. For example, using executable assertions while performing structural or functional testing can provide a great deal of assistance in error detection and location. The next report will provide guidelines for combining the individual techniques into an effective test strategy.

The improvements described below do not change the relative rankings of the test techniques as presented in the previous sections. We anticipate that all of the techniques will be improved in the future. Many static analysis checks will be incorporated into the compilers that implement advanced programming languages--this will make static testing more automatic and universally used. Dynamic testing methods will be better defined so that they will be commonly understood and applied by practicing programmers. Successes with the use of formal techniques will encourage researchers to develop ways of making them more practical.

7.1 STATIC ANALYSIS

Extraneous warning messages are the biggest complaints of users of static analysis. Two types of anomaly checking account for most of these "red herrings": mixed-mode arithmetic and data flow analyses. The number of extraneous messages can be significant: a 5000-statement FORTRAN program tested by Gannon, et al.[1] using GRC's SQLAB tool produced about 130 mixed-mode arithmetic warnings, none of which indicated improper program operation. The same program was seeded with 37 errors and processed by the DAVE static analysis system; it generated 580 data flow error and warning diagnostic messages.[2]

Anomaly detection will inevitably generate false alarms, because its purpose is to find programming practices that are capable of producing errors but not certain to do so. In some situations programmers find it convenient to violate a principle enforced by anomaly detection. For example, most FORTRAN compilers provide automatic conversion from integer to real and real to integer; so programmers do not bother to reference the IFIX and FLOAT routines.

Most static analysis tools (including SQLAB and DAVE) give the user the option of suppressing individual anomaly detection messages. Of course, the danger in this is that actual errors can go undetected. Probably the best solution to the mixed-mode warning problem is to use a strongly-typed language or a compiler that forbids implicit type conversions. If a programmer knows that he has to write out every conversion for the compiler to accept his code, he will be careful to do so. Hopefully, as he codes the type conversions he will mentally check that they are proper. The permissive FORTRAN compiler tempts the

[1] C. Gannon, R. N. Meeson, N. B. Brooks, An Experimental Evaluation of Software Testing, General Research Corporation CR-1-854, May 1979, p. 5-8.

[2] This data is derived from the author's examination of an experiment performed by Carolyn Gannon. Not all the messages were extraneous, but this is at least 16 messages per error found.

programmer to write implicit conversions, turn off the static mixed mode messages, and hope nothing goes wrong.

Data flow anomalies present a different problem, since they are possible with any programming language or compiler. There are two major causes of extraneous data flow messages: global entities and unexecutable paths.

Advanced static test tools such as DAVE perform a careful analysis of global data flow. However, they can be stymied by code which references system or library routines that the tool has no information about. The user can supply this information by providing "stubs" - skeleton versions of routines that contain no executable code but indicate parameter usage. The SQLAB tool facilitates the use of stubs by helping the user to develop and save a library of stub modules.[1] Stubs of tested modules can be built automatically and put on the library; routines that reference these modules can then be analyzed for global data flow without including large amounts of additional source code.

The developers of DAVE estimate that 15% of all error and anomaly messages produced by the tool are tied to unexecutable paths.[2] As discussed in Sec. 5.3.3 , the general problem of determining whether a path can be executed is unsolvable. However, researchers are looking for algorithms that will detect certain unexecutable sequences that frequently arise in practice.[3] Such algorithms would also be useful for test data generation in structural testing.

---

[1] S. H. Saib, et al., Advanced Software Quality Assurance Final Report, General Research Corporation CR-3-770, May 1978, pp. 143-144.

[2] L. J. Osterweil, "The Detection of Unexecutable Program Paths Through Static Data Flow Analysis", COMPSAC '77, Chicago, Nov. 1977, p. 411.

[3] Some of this research is briefly summarized in H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil, "On Two Problems in the Generation of Program Test Paths," IEEE Transactions on Software Engineering, Vol. SE-2, No. 3 (September 1976), p. 227. The article by Osterweil cited above relates the unexecutable path problem to static testing.

7.2 EXECUTABLE ASSERTIONS

The mechanics of assertion testing are very simple. Assertion testing is harder than static testing because of the analytical work required to develop a complete and effective set of assertions, and because assertions may not pinpoint the locations of errors.[1] To make his job easier, a tester needs information about how to formulate and where to put assertions.

Therefore, assertion testing needs to be improved in two areas. First, a methodology of assertion testing should be developed to provide guidelines for putting assertions in programs. Second, test tools should be able to indicate specific sections of code that need more assertions. Changes and expansions of the capabilities of assertion languages may accompany these improvements.

One method of forming assertions is to perform a formal verification of a program using the Floyd method.[2] Of course, formal verification is a very time-consuming process and may not be part of the planned test effort. However, methods of generating assertions can be carried over from proving to testing. The application to testing of some formal assertion generation techniques is described by Laski.[3]

[1] If assertion testing were used by itself, the user would also have to formulate his own test input data. In reality, structural and functional testing methods are available, and we deal with their problems in the following subsections.

[2] R. L. London, "Perspectives on Program Verification", in Current Trends in Programming Methodology, Vol. II (R.T. Yeh, ed.), c. 1977, Prentice-Hall, pp. 151-172.

[3] J. W. Laski, "A Hierarchical Approach to Program Testing", SIGPLAN Notices, Vol. 15, No. 1 (January 1980), pp. 77-85.

Other researchers have taken a less rigorous but more practical approach to developing an assertion methodology. Three papers have recently appeared which have defined specific goals of assertion testing and outlined assertion constructs which support these goals.

From these papers a composite picture emerges of how assertions can be used to test programs. Candidate locations and purposes of assertion checks include:

- The entry points of a module, to check values of incoming parameters
- Input statements, to ensure that meaningful values are accessed
- Following a call to another module, to be sure that values returned are acceptable
- At each control point (branching statement, loop beginning and termination) to check that conditions are compatible with the path taken
- After complex computations, to prevent propagation of an error[1]
- To impose conditions that are required to hold over entire sections of code (e.g. a loop or module)
- To check histories of computations, which involves comparing the current value of a variable against its previous values[2]

[1] The first five are due to D, M. Andrews, "Software Fault Tolerance Through Executable Assertions", Twelfth Annual Asilomar Conference on Circuits, Systems, and Computers, Nov. 6-8, 1978, Pacific Grove, California.

[2] These two are due to R. N. Taylor, "Assertions in Programming Languages", SIGPLAN Notices, Vol. 15, No. 1 (January 1980), pp. 105-114.

- To provide safeguards on the integrity of data structures (e.g., pointers, counts, value bounds)[1]

The adoption of such a specific list of purposes for assertions could expedite testing in several ways. Static analysis can identify many of the candidate locations noted above; static tools could produce concise reports that contained the information needed to formulate the simpler assertions. Other aids could be made available for types of assertions that require greater judgment, such as the last four in the list. Possible sources of information for the more difficult assertion types include complexity metrics, formal verification algorithms, and requirements and design documents.

Extensions to current assertion languages can make it easier to express assertions. Some advanced assertion constructs have been suggested by Chow.[2] These include:

- The ability to selectively activate or deactivate individual assertions without recompilation
- Global assertions, which must hold over entire modules
- The ability to specify the execution sequence or some of its properties in an assertion predicate
- The ability to reference previous values of variables

[1]The last is due to S. S. Yau, J. L. Ramey, R. A. Nicholl, "Assertion Techniques for Dynamic Monitoring of Linear List Data Structures", Journal of Systems and Software, Vol. 1, No. 4 (1980), pp. 319-336.

[2]T. S. Chow, "A Generalized Assertion Language", Proceedings - 2nd International Conference on Software Engineering, San Francisco, October 1976, pp. 392-399.

7.3 STRUCTURAL TESTING

The two major problems with structural testing are error detection and test data generation. Structural testing by itself provides no mechanism for error detection at all--the user must supply this through some other, usually manual, means. As for test data generation, the various tools that support structural testing provide indications of how the level of coverage might be increased, but the user must ultimately derive the test cases himself.

These shortcomings are mitigated by the fact that structural testing is not conducted in a "vacuum"—that is, without the benefit of other techniques and sources of information. We have emphasized that structural and functional testing should always be combined to ensure the effectiveness of either technique. There are great benefits to be gained from integrating several other test techniques with structural testing.

An obvious way to provide automatic assistance in error detection is to use executable assertions. Assertions are better than the execution traces produced by instrumentation tools for two reasons. First, assertions express relationships between variables rather than just reporting their values—in this way error conditions can be checked automatically. Second, execution traces tend to produce large amounts of output, which is wasteful of computer resources and annoying to the user.

Symbolic execution techniques have been applied to the problem of generating test data to cause a particular section of code to be executed. The problem of determining the necessary conditions for such data is equivalent to finding the predicate conditions, in terms of input variables, for the entry point to that section of code. Clarke[1]

[1] L.A. Clarke, "Automatic Test Data Selection Techniques", <u>Infotech State of the Art Report--Software Testing</u>, Infotech International, Berkshire, England, Vol. 2, pp. 43-63 (1979).

describes a tool which uses symbolic execution to generate test data to satisfy the statement, branch, and path coverage criteria.

Test data generation based on symbolic execution is not yet available in a tool suitable for general use. A more limited but practical aid for structural testing is a "test assistance" report, described by Deutsch[1] for the RXVP80™ test tool.[2] The test assistance report associates unexecuted branches with the decision statements that control their execution. A listing of the statements affecting the values of the variables forming each predicate is also produced. The user must decide how to change the program inputs to cause a change in the value of the predicate.

7.4 FUNCTIONAL TESTING

When Howden[3] first introduced functional testing he was somewhat vague about the mechanics of the technique. Recently he has described how programs can be decomposed into functional parts from design documents.[4] Others,[5,6] have been working on ways to identify special values for test cases. Unfortunately, no one has tried to develop an automated tool to provide general support for functional testing.

[1] M. S. Deutsch, "Software Project Verification and Validation", Computer, April 1981, pp. 64-66.

[2] RXVP80™ is a test tool built and marketed by the Software Workshop™, General Research Corporation, Santa Barbara, California.

[3] W. E. Howden, "Functional Program Testing", IEEE Transactions on Software Engineering, Vol. SE-6, No. 2 (March 1980), pp. 162-170.

[4] W. E. Howden, "Functional Testing and Design Abstractions", Journal of Systems and Software, Vol. 1, No. 4 (1980), pp. 307-313.

[5] L. J. White, E. I. Cohen, " A Domain Strategy for Computer Program Testing", Infotech State of the Art Report - Software Testing, Infotech International, Berkshire, England, Vol. 2, pp. 325-363 (1979).

[6] K. A. Foster, "Error Sensitive Test Cases Analysis (ESTCA)", IEEE Transactions on Software Engineering, Vol SE-6, No. 3 (May 1980), pp. 258-264.

As in structural testing, error detection and test data generation cannot be completely automated in functional testing. Furthermore, to effectively conduct functional testing, a tester must have a good understanding of the way a program is designed and the tasks that it is to perform. Automated tools cannot provide substitutes for this knowledge.

But an automated tool could support functional testing in various ways. It could relieve the tester of clerical burdens. It could make it easier to test sections of code separately. It could promote thorough testing by tabulating tests that have already been run and identifying input regions that have not been explored.

These capabilities exist individually in several existing tools that were developed for software testing. The "Adaptive Tester"[1] developed by GRC provides automatic assistance in testing large software systems. It has extensive data analysis and reduction algorithms to handle large amounts of test output data. It generates several types of graphic displays of data points in any selected combination of input- and output-space. These displays would permit White and Cohen's domain testing strategy to be implemented by selecting test data with a cursor.

A conventional test harness[2] can help to separately test functionally distinct sections of a program. The tester must determine the separation, but then the tool permits him to monitor and control execution. The tool can drive sections of code without the need for a main program; it can identify external references that need "stubs" provided for them; it can display the values of variables at intermediate points in the computations.

[1]C. G. Davis, "Testing Large, Real-Time Software Systems", Infotech State of the Art Report - Software Testing, Infotech International, Berkshire, England, Vol. 2, pp. 85-105.

[2]For example, the TPL system described in D.G. Panzl, "Automatic Software Test Drivers", Computer, Vol. 11, No. 4 (April 1978), pp. 44-50.

A test tool which combined the features needed for functional testing would be very useful. It would also help the technique of functional testing to mature.

7.5 FORMAL TECHNIQUES

Researchers in symbolic execution and formal verification are beginning to respond to the challenge of producing automated tools suitable for general use. But there are many obstacles to the acceptance of formal techniques by the general programming community. We feel that it will be a long time[1] before highly automated formal verification "packages" will be in widespread use.

However, we expect a great deal of progress to continue to be made in three areas concerning the application of formal techniques. First, applications of formal methods to testing (as opposed to program proving) will be refined. Second, tools will be built which make formal verification much easier for the specialists themselves. Third, working programmers will begin to understand and believe in formal verification.

Much has been written about applying algorithms derived for formal techniques in static and dynamic testing. Earlier in this section we discussed the use of symbolic execution in static data flow analysis and in test case generation for branch and path testing. Some static analysis tools (including AMPIC, DAVE, and SQLAB) make use of algorithms similar to symbolic execution to perform data flow analysis and reaching

[1]Susan Gerhart makes the following statement: "... optimistic projections for full mastery of the type of theorem proving we want today, namely interactive guidance by user-supplied strategies through fully mechanized subproofs, are ten years, with full capabilities for finding proofs, although not necessarily finding interesting theorems, in thirty years." (emphasis hers). S. L. Gerhart, Program Verification in the 1980's: Problems, Perspectives, and Opportunities, University of Southern California/Information Sciences Institute, Report No. ISI/RR-78-71, August 1978.

set generation and to detect infinite loops and dead code. We expect the research in formal techniques to provide more ideas for improving other test techniques.

Recent formal verification efforts have made researchers aware of the awkwardness of current methods. Walker, et al. drew the following conclusion while verifying the security of an operating system kernel:

> "Improvements in the theorem prover's power and user interface would of course be valuable. However, mechanisms are also needed to minimize unnecessary theorem prover invocations, which result either from redundant proofs or reverification."[1]

Gerhart[2] suggests several ways to streamline the proof process, such as using more flexible approaches to proof organization, use of multiple theorem provers in parallel, and maintaining libraries of proofs. Moriconi[3] has been working on the reverification and incremental verification problems. Ideas for improving verification tools will continue to grow out of efforts to prove programs.

Probably the most important problem that formal verification researchers must solve is how to convince skeptics in the software community of the value of their work. The recent successes in verifying the security properties of several operating systems have raised the morale of the researchers themselves. However, they are aware

---

[1] B. J. Walker, R. A. Kemmerer, G. J. Popek, "Specification and Verification of the UCLA Unix Security Kernel", Communications of the ACM, Vol. 23, No. 2 (February 1980) p. 127.

[2] Gerhart (op.cit.), p. 7,11.

[3] M. Moriconi, "Toward Incremental and Language-Independent Program Verification Systems", Verification Workshop (VERkshop) Proceedings, SRI International, Menlo Park, California, April 21-23, 1980.

that many in the outside world believe formal verification to be tedious, arcane, and generally not worth the trouble.

Proponents of formal verification hope that several developments will help them overcome the skeptics. When a system that has been subjected to verification is put into everyday use, the argument that formal techniques cannot be applied to real programs will disappear. It is even more important that formal verification become "de-mystified". Verifiers need to learn how to communicate their proofs to an audience of programmers. They also need to teach programmers how to do formal verification.[1]

[1]These comments are summarized from Gerhart (op.cit.). Undergraduates at many universities are now learning formal verification methods; some are also using symbolic execution systems and theorem provers.

8 AREAS FOR FURTHER RESEARCH AND INVESTIGATION

We have identified six topics concerning testing in which further research is needed. Four of these topics fall into the category of studies or experiments that should be performed to fill in gaps in the available knowledge and data about testing. The other two topics concern new types of test tools that should be investigated and built.

8.1 STUDIES AND EXPERIMENTS

The topics for future studies and experiments are:

- The costs of using the test techniques described in this report
- A methodology for assertion testing
- A methodology for functional testing
- An error classification system useful for evaluating test techniques

There is an unfortunate scarcity of data on the costs of using the test techniques described in this report. There is almost no data on the amount of analysis time required by each technique. Efforts should be made to monitor the time and costs involved in using the test techniques during several software development projects. This data will make it possible to compare costs between the techniques, and between automated techniques and manual testing.

Executable assertions have too long remained a testing concept rather than a method of testing. Assertions are not widely used by software developers, not because they are not understood or are looked upon unfavorably, but because they have not been incorporated into a standard programming and testing methodology. Programmers want to know what types of assertions are most useful and how to go about writing them before they will be more willing to use them.

The technique of functional testing also needs quite a bit of refinement to become a standard methodology. All software developers currently use some of the concepts of functional testing (although they may not use that term), but they are applied in an ad hoc rather than systematic manner. Software developers need to be made aware of the automated aids which are available for performing data reduction and stress testing. Techniques for decomposing programs into their functional components need to be developed.

In Sec. 3.1 we discussed the inadequacies of the TRW error classification system for evaluating the effectiveness of testing. An error classification system which clearly distinguishes between instances and symptoms of errors should be developed. The symptomatic categories should reflect the ways that the various test techniques actually detect errors.

8.2 NEW TEST TECHNIQUES AND TOOLS

Two ideas for new test tools are the following:

- An integrated functional testing tool
- An advanced assertion preprocessing aid

In Sec. 5.4.1 we identified several existing types of test tools which support various activities of functional testing. A most useful tool would have a combination of these capabilities which could be used together. Such a tool should provide test harness and test driver capabilities, so that partially completed systems could be tested regardless of whether top-down or bottom-up development was used. The tool should have a test library facility so that tests could easily be repeated and stored. Advanced mathematical techniques for data reduction and stress testing should also be available for making and analyzing large numbers of tests.

In Sec. 5.2.1 we noted that available assertion preprocessors do not incorporate all desirable assertion constructs. Furthermore, there are many combinations of computer system and programming language for which assertion preprocessors are not available. The possibility of using recent advances in translator writing tools to build assertion preprocessors should be investigated. Such a tool might automatically generate an assertion preprocessor from syntactic and semantic descriptions of the assertion constructs and the target language.

PRECEDING PAGE B

APPENDIX A
GUIDELINES FOR TESTING SOFTWARE

CONTENTS

PRECEDING PAGE B

A.1 INTRODUCTION

This appendix describes guidelines for testing software systems. It is not a complete handbook for testing software but attempts to give some guidance in the use of techniques, tools, and approaches. Testing software is a very labor-intensive and creative act. The person testing the software requires insight to decide how to test it and how to interpret the testing results. He also must decide how much testing must be done in order to conclude that the software is running correctly.

These guidelines are presented as a set of hints in organizing the testing process. They have been developed by studying the current ways in which software is being tested, by doing research into new methods of testing, and by drawing on our experience in testing large software systems. They are by no means the complete and the final word on how to test a software system. More research and experimentation needs to be done. The techniques described in these guidelines must be applied in projects to discover their limitations and to improve the ways in which they are applied.

This set of guidelines for testing software aims to help the tester achieve the following:

- Understand what attributes of the software to test.

- Know which analysis tools and testing techniques to use.

- Understand the benefits of applying the analysis tools and testing techniques.

- Learn a testing methodology which provides an incremental approach to testing with the tools and techniques.

- Know when to begin testing and when to stop testing.

- Measure how thoroughly the software has been tested when the tools and techniques have been applied.

- Establish a level of confidence in the software.

A.2 TESTING SOFTWARE SYSTEMS

Testing a software system is different from testing an individual program. In The Mythical Man-Month[1], Brooks defines the difference between a program and a "programming systems product". A program is complete in itself. It can be tested by itself and it does not interact with other programs or devices. It is run by the person who wrote it on the computer on which it was written. A program can become complicated in two ways: it can become a programming product, or it can become part of a programming system.

A programming product is a program which has been generalized. It may perform one task or implement an algorithm, but its range of inputs and the situations in which it can be used have expanded. It can be run on computers other than the one on which it was developed. It requires good documentation so that it can be used by people who did not write the program. Finally, it needs to be thoroughly tested, since the users of the program may not have the knowledge or resources to correct any errors that they find in the program.

[1] F. P. Brooks, Jr., The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley, Reading, Mass., 1975.

A program can also become part of a programming system. It now must interact with other programs to perform a task. In order to interact correctly and achieve the task, the program must adhere to rigidly defined interfaces. It also must adhere to constraints on the way in which it will operate including running time, memory requirements, the input and output devices it uses, and the way in which it accesses and uses global information. The program must now be tested in concert with the other programs; this increases the time and cost of testing.

A programming systems product is a program with both sets of characteristics. It is intended to run in several different environments together with other programs. It may interact with other systems or be part of a larger system (an embedded programming system). Brooks estimates that this type of program costs nine times as much to develop as a simple program. Consequently, it also may cost nine times as much or more to test.

Problems in Testing Programming Systems

Programming systems are difficult to develop and test because:

- The interactions between the individual parts become complex.
- The size and number of programs involved increase dramatically.
- The amount of effort that must be expended to manage and understand a large system increases also.

One of the most difficult problems in building and testing a software system is managing the volume of information. Tools and techniques which are adequate for testing a small set of programs quickly become inadequate for testing a large set of programs. Tools and techniques have to be designed primarily to handle large amounts of programs and data. The data taken in testing a system must also be managed, and this may require another tool.

Independent Testing

When systems are built, testing which in the case of the individual programs was done by the developer, now becomes a task best performed by someone else. Since the system is a product which will be used by a number of different users, it is best tested by someone who understands how it should be used but who did not develop it. Because the program is now a system, it needs to be tested by someone who has an understanding of the overall architecture of the system and how the parts are to interact and communicate. The individual developers of the programs in the system may not have this knowledge. Once again, it is better to have an independent but knowledgeable tester.

Standards

One of the ways to handle complexity is standardization. If systems can be constructed out of similar, well understood parts, then the system can be better understood as a whole. If every program in the system has a similar format, uses the same set of control structures, and adheres to the same interface standards, then any program in the system can be more easily understood. Each program complies with a standard set of assumptions.

Testing techniques and tools can also take advantage of the common structure and assumptions. Tools can use the assumptions to become more efficient. They can also test the programs to see if they follow the standards. And if new tools need to be developed for a special application, then they can take advantage of the similar structure and assumptions and be quick and easy to implement.

Documentation and Organization

There have been a number of standards adopted for testing computer software.[1] Usually these standards set down requirements for documents, formats for documents, and schedules and topics for formal reviews during the software development process. Although documents and reviews are important in the development and testing of software, they are not the main subject of this apppendix. Documents and reviews support and give structure to the development and testing tasks but they do not define how they should be done and what tools and techniques should be applied.

A.3 WHAT TESTING ENTAILS

The first question that a tester asks is "How am I going to test all these programs?" Implicit in this question are two other questions: (1) What do I have to test for? (2) What can I use to help me test for it? Testing requirements answer the question of what to test for and tools and techniques answer the question of what to use for testing.

A.3.1 Testing Requirements

When a tester first starts thinking about what to test, he thinks about testing what the program does. That is, he first tests its functions. Later, he may also consider special cases such as boundary conditions and individual values which are somehow important. If the program is part of a system, he also may test its performance. That is: how much memory it uses, how long it runs, or how many data values it can handle in a certain amount of time. However, there are other

[1] E. A. Straker, C. E. Fenner, J. Penland, and T. E. Albert, A Methodology for the Validation of Real-Time Software Used in Nuclear Plant Safety Applications, Science Applications, Inc. Report No. SAI-78-517-LJ, April 1978.

important features of the software which also must be tested. These features can be grouped under the title of "testing requirements" and include the following:

- Functional requirements
- Performance requirements
- Currently accepted programming practices
- Semantic rules of the programming language being used
- Constraints placed on the programs by specifications, other programs, the hardware, and the operating environment
- Assumptions made by the designers and programmers in implementing the system

Testing requirements can also include all the hardware, software, people, and other things involved in performing a test. Military standards[1] describe documents for defining and running tests but we will not be concerned with those in this report. In this context, testing requirements will mean all those things that a tester should consider in designing a set of tests for a software system.

Currently Accepted Programming Practices

The most well-known currently accepted programming practice is structured programming. Structured programming restricts the types of control structures that are allowed in the programming language. It also may be combined with a design methodology such as top-down design

[1]"Testing of Computer Programs", AFR 800-14, Volume II, Chapter 5.

or levels of abstraction which serve as guidelines for the software design and development process. Current accepted practices may include other things such as requirements for comments in certain places in the code, a standard program organization, and certain formatting conventions. They all have one thing in common. They make the program more easy to read, maintain, and understand. They are rules of thumb, not absolute, provable laws, but they are accepted by the software industry as a whole. They are important in the testing process if a quality piece of software is to result.

Programming Language Semantic Rules

Each statement in a programming language has a meaning attached to it. Its meaning is most explicitly expressed as the set of machine language statements it translates into. Its meaning may also be expressed by how it alters or keeps constant the state of the machine it is running on. Programming language semantics have also been expressed formally[1]. Sequences of programming language statements may also have meanings attached to them in addition to those attached to the individual statements. In PASCAL, for example, a particular ordering of the statements in a program is prescribed. Certain statements, such as the IF statement, may require a matching statement, such as an ENDIF statement, to define the end of a sequence.

Not all of these semantic constraints on the programming language are explicitly defined in the programming language, and not all are checked by the compiler that processes the language. For example, some FORTRAN compilers do not check the data types of parameters to subprograms. Failing to adhere to these constraints can cause software to operate incorrectly. Many of the checks for these constraints have been

[1]M. J. C. Gordon, The Denotational Description of Programming Languages, An Introduction, Springer-Verlag, 1979.

embodied in static analysis tools. This is the most common way in which software is checked for violation of these constraints.

Constraints

A program must also adhere to the constraints placed upon it by the enviroment in which it runs in and the underlying theory from which it was developed. For example, a real-time control program must take into account the time constants of the devices and sensors with which it interacts. Programs which compute orbits of satellites must conform to the underlying theory of orbital mechanics. These constraints and underlying theories are a source of many software errors. Therefore they are also a good source for test cases. The constraints are an especially good source for many of the assertions used in assertion testing and in formal verification.

Programmer's Assumptions

There are also many assumptions made by programmers in designing and implementing programs which are not part of the formal design process. These are a frequent source of errors. They include assumptions about the range of values of variables, special quirks of the compiler or operating system being used, and data base values which are set before the program is run. All these assumptions need to be made explicit and tested for their effect on the system as a whole.

A.3.2 A Testing Methodology

A good testing methodology must include all of the following in order to test software well:

- A set of programming standards
- A test plan
- A set of testing techniques
- A standard of performance for the software

__Programming Standards__

A set of programming standards must be defined if testing is to be successful. These standards should include:

- The currently accepted programming practices to be followed on the project

- A format for the text of each program

- Requirements for specific kinds of comments in specific places in the program

- Constraints on the use of the programming language, operating system, and command language of the computer

__Test Plan__

A *plan for testing* the software must be developed. This plan should include:

- An ordered sequence to testing so that groups of modules are tested together before the whole system is tested

- A set of data values and scenarios to be used in the testing

- Methods for comparing the output resulting from the testing to the expected output

The test plan should also include ways in which the functional and performance requirements of the software will be tested and how the other standards, constraints, and assumptions will be verified.

Testing Techniques

The testing methodology must also specify a set of testing techniques which will be used to accomplish the test plan and when they will be used. It should specify which techniques should be used during each phase of development and in each phase of the testing. It also must specify which techniques will be used to evaluate how completely the software has been tested.

Standards of Performance

The testing methodology must also specify what method will be used to decide when to stop testing. It must describe how to determine when the software has attained a specified value of reliability or functionality which indicates that it can be accepted. This requires keeping track of the testing history and the development process. This in turn requires keeping a data base, possibly a tool for maintaining the data base, and a tool for measuring the reliabilty value.

A.4 APPLYING TESTING TOOLS

The test techniques to be considered here, and the tools that implement them, were discussed in detail in the body of this report. The five techniques which we consider are:

- Static analysis
- Executable assertions
- Structural testing
- Functional testing
- Formal techniques--symbolic execution, formal verification

A.4.1 Relating Tools to Software Development Phase

Testing is not confined to one phase of a project; rather it begins as soon as programmers begin to sift the mistakes out of their code and continues as long as the final product is in use. Table A.1 presents the phases of the software development life-cycle and notes the test-related activities for each phase. The phases and activities may overlap in time.

Testing should begin as early as possible in the software life-cycle. Boehm's study[1] relating life-cycle phase to repair costs reported that it is about 15 times more expensive to fix an error in the maintenance phase than it is during coding. However, errors will probably always appear thoughout the life-cycle, so the evaluation of test techniques cannot be focused on one particular phase.

Different test techniques lend themselves more readily to different life-cycle phases. For example, static analysis techniques can be applied very early in the coding process, while most dynamic testing techniques must wait until a compiled *version of* at least one complete module is available. Table A.2 shows a matrix of test techniques along with the life-cycle phases in which their application may be appropriate.

It should be emphasized that using a technique later in the life-cycle does not imply lower cost-effectiveness. Different techniques catch different kinds of errors, and the effects of different errors on program operation are rarely the same. The conclusion to be drawn from Boehm's "increasing cost of error correction" data is that there is a penalty to be paid for delaying the testing process.

[1] B. W. Boehm, "Software Engineering," IEEE Transactions on Computers, Vol. C-25, No. 12 (December 1976), pp. 1226-1241.

TABLE A.1
LIFE-CYCLE PHASES AND TESTING ACTIVITIES

Phase	Definition*	Testing Activities†
Analysis	Define functional performance requirements, devise and analyze development plan alternatives.	Determine testing objectives.
Design	Establish details of program components and system configuration.	Formulate test strategies. Plan for development or acquisition of test tool.
Coding and Checkout	Program individual modules.	Insert assertions during coding. Use static analysis to get "clean compiles."
Test and Integration	Unify modules into complete system. Test program against requirements.	Determine test plans. Perform module (unit) testing, and preliminary and formal qualification testing.
Installation	Development Test and Evaluation of system. Adapt program to operational site (if necessary).	Test system in its operational environment.
Operation and Support	Maintain and modify software.	Use tests as diagnostics for error correction. Repeat above lifecycle activities when implementing modifications.

*Software Acquisition Management Guidebook: Life Cycle Events, USAF Electronic Systems Division, ESD-TR-77-22, prepared by The MITRE Corporation, February 1977, pp. 49-52.

†Acquisition and Support Procedures for Computer Resources in Systems, AFR 800-14, Vol. II, 26 September 1975, pp. 5-1 through 5.5.

TABLE A.2
APPLICATION OF TEST TECHNIQUES IN THE LIFE CYCLE

	Life Cycle Phases (see Table A.1 for descriptions)			
Techniques	Coding and Checkout	Test and Integration	Installation	Operation and Support *
A. Static Analysis				
1. Program error detection	✓	✓		✓
2. Anomaly detection	✓	✓		✓
3. Assertion checking	✓	✓		✓
4. Test data generation		✓		
5. Automated documentation	✓	✓		
B. Dynamic Testing				
1. Testing aids				
a. Executable assertions		✓	✓	✓
b. Functional testing		✓	✓	✓
c. Instrumentation		✓		
d. Structural testing		✓		
e. Test harness		✓		
2. Debugging aids		✓		✓
3. Performance evaluation		✓	✓	
C. Formal techniques				
1. Symbolic execution		✓		
2. Formal verification		✓		

*This column denotes error diagnostic testing only. Modifications to operating programs must undergo testing as for the previous phases.

Guidelines for testing software systems are summarized in Table A.3. They show the steps to be considered in test planning, and the correspondence between test requirements and techniques.

TABLE A.3

SOFTWARE TESTING GUIDELINES

STEP 1. Estimate the number of errors in the software, using either of the following techniques:

a. Error seeding.
b. Halstead effort metric.

STEP 2. Use the techniques to evaluate the software on the following testing requirements.

Technique	Requirement
Code review	Current accepted programming practices
Static analysis	Programming language semantic rules
Structural testing	Implemented structure (will include some functions)
Assertion testing	Some functions, requirements and constraints
Functional testing	Implemented functions
Symbolic execution	All implemented functions
Formal verification	All software requirements and all assumptions

STEP 3. Apply the techniques in the order indicated in Table A.4.

STEP 4. Measure the reliability of the program using a reliability metric.

STEP 5. Stop when the number of errors found reaches the number of errors estimated, and when the reliability of the program is satisfactory.

A.4.2 Making an Error Estimate

The first step in the testing process is to estimate how many errors can be expected to be found in the software. This is done to give an estimate of how long testing should continue. This approach has not been used before; but we recommend it because it can give a quantitative answer to the question of when to stop testing.

An error estimate can be made in several different ways. Two discussed in Sec. 3.2.4 of this report are the error seeding technique and the Halstead effort metric. Error seeding can be used in conjunction with a formula presented in Schick and Wolverton[1] to give an estimate of the total number of errors in the program at any point in the testing process. The Halstead effort metric[2] has been shown by Fitzsimmons and Love[3] to correlate with the number of errors present in a program at the start of the testing cycle.

The error estimate should not be regarded as exact, but rather as a "ballpark figure" which is useful for planning purposes. If possible, the estimate should be refined to reflect the characteristics of the software development environment: the skills of the people working on the project, the nature of the application area, the computer and language being used, etc. We feel that people will become able to make fairly accurate error estimates as they gain experience with them.

[1]G. J. Schick and R. W. Wolverton, "An Analysis of Competing Software Reliability Models," IEEE Transactions on Software Engineering, Vol. SE-4, No. 2, (March 1978), pp.112-114.

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

[3]A. Fitzsimmons and T. Love, "A Review and Evaluation of Software Science", ACM Computing Surveys, Vol. 10, No. 1 (March 1978), pp. 3-18.

However, the error estimate should not be the only guideline used in gauging the magnitude of a test effort or in applying the test techniques. The most important criterion for testing is the performance of the product under test, not the number of errors found.

A.4.3 Test Technique Selection

Chapter 5 of this report contains characteristic profiles of the tools and techniques. It indicates which errors each technique will identify, how effective and reliable each technique is in locating errors, and estimates the cost of using each technique. Each technique is useful in testing software on one or several of the testing requirements. The tester should select a number of techniques based on the cost he expects to encounter during testing, the number of errors estimated to be in the software, the reliability that must be achieved, and the availability of the tools.

The testing techniques and the testing requirements that they address were shown in Table A.3. In general, techniques should be selected in the order in which they appear in that table. That is, structural testing should not be done unless code reviews and static analysis are done. This is because techniques occurring earlier in the table are usually less costly to apply and are more automated.

All techniques are more effective and easier to use if some restrictions are placed on the code during its design and development. Standards are important. You must decide how you are going to test the software BEFORE it is designed and built. Most tools and techniques make assumptions about the structure of the system being tested. They also are less costly to use if some constraints are placed on the software. In addition, if new tools must be built, they can be developed easier and more quickly if they can make some assumptions about the code that they have to process.

The most basic testing technique is code review. This can be done either by hand or automatically. Code review checks the constraints and standards that were set down for the software development. Standards for structured programming, program layout and style, comments, and programming language restrictions are checked by this technique. If only one technique is used besides testing the basic functions of the software, this is the one that should be used. It is the only way to verify that the software adheres to the standards set down for it.

Static analysis can be used to verify that the code adheres to the constraints of the programming language. Here such things as interfaces, mixed-mode arithmetic, and data flow can be checked.

Structural testing is the next technique to be included. This can usually be accomplished at the same time that the basic function of the software is being tested. Instrumentation can be placed in the code and the amount of code exercised by the basic tests can be measured. New tests can then be derived which exercise greater amounts of the software.

If assertions have been placed in the code during the development, they can be used during the basic functional testing to verify that the software follows the assumptions and constraints placed upon it. If needed, more assertions can be added to the code to check new conditions that are imposed as the software is being developed and to check for and recover from error conditions.

Given that the other techniques have been used, full functional testing can be attempted in order to verify the complete function implemented by the software. This means testing for special input values, stress testing, and testing the domains associated with each input variable. The other techniques -- structural testing and assertion testing — can be used to give information by which to evaluate the results of functional testing.

If the reliability requirements of the software are high, then symbolic execution can be used either to generate test cases for the software or to summarize its function. Symbolic execution is more effective if assertions have already been placed in the code. The decision to use symbolic execution must be made when the software is designed. This decision can help reduce the complexity of the code and make the logical expressions developed by symbolic execution easier to generate and interpret.

Finally, after all other testing methods have been used and if the software has been designed with verification in mind, then program proof can be attempted. The goal of the proof may be to show that certain sections of the software are correct, that they terminate or that they implement a secure piece of software. In any case, this is the most difficult and expensive technique. It should only be used when the requirements for correctness or security are high and only then if the software has been designed with proof in mind.

A.4.4 Order of Applying the Techniques

The testing techniques should be applied in the order shown in Table A.4. If structural testing and assertion testing have been decided upon, then the assertions and the ability to trigger instrumentation should be added during development of the code. Likewise, the programming standards and conventions should be followed throughout the development process.

Notice that formal verification must be preceded by the other test techniques. This is because code cannot be verified unless it is correct. Coding errors can be corrected during the process of conducting code-level proofs, but this is an inefficient way to operate.

TABLE A.4

ORDER OF APPLYING TESTING TECHNIQUES

1. Insert executable assertions during coding of the program.
2. Perform code review and standards checking, using automated tools if available.
3. Perform static analysis.
4. Perform programmer-defined tests, unit tests, or integration tests and record structural coverage and assertion violations.
5. Use structural tests to get complete coverage of program branches.
6. Perform full functional testing.
7. (Optionally) Perform symbolic execution to generate test cases or make functions explicit.
8. (Optionally) Attempt formal verification.

Formal verification of the design of the software may or may not rely on the code level proofs[1]; but the code should be written and tested before the design-level proofs are conducted. This is because problems encountered during coding and testing may require changes in the design.

[1] B. J. Walker, R. A. Kemmerer, and G. J. Popek, "Specification and Verification of the UCLA Unix Security Kernel", Communications of the ACM, Vol. 23, No. 2 (Feb. 1980), pp. 118-131.

A.4.5 Levels of Confidence in Testing

Since testing is done to improve the quality of a piece of software, it is natural to look for a way to measure the "amount" of quality improvement that results from following a particular test plan. Unfortunately, there are no general measures of software quality that are suitable for use in "measuring the benefit" of testing.

However, there is a natural progression of software quality "levels" that is followed when software is tested. Each level describes the state of development of the software, and reflects the confidence one can have that the software will perform properly. The levels represent goals to be achieved during the testing and verification of software. As such, these levels fall within the general framework of software development goals or milestones, such as "code compiles successfully" or "installation successfully completed on the target machine".

The ten levels as we identify them are as follows:

1. The code complies with the set of programming standards mandated for it.

2. The interfaces between all program units are compatible.

3. The software performs satisfactorily for an initial set of test cases defined by the programmers (checkout tests).

4. All sections of code have been exercised by at least one test case. The output for these cases is correct.

5. The software accepts arbitrary inputs "gracefully"—that is, no input conditions can cause undesirable behavior such as an abnormal program termination.

6. The software produces correct output for a set of special-values functional tests (see Sec. 5.4.1).

7. The software produces correct output for a set of functional stress tests (see Sec. 5.4.1).

8. The intermediate states of computation are correct for the tests described in (3), (4), (6), and (7).

9. The coded software faithfully implements the design.

10. The software always produces acceptable output that is in accordance with its specifications.

If the test plans described in these guidelines are followed, then these quality levels will be achieved. Traditional testing--manual code inspection and executing programmer-defined tests--can only achieve the first three levels. The static and dynamic test techniques described in this report cover the first eight levels, while formal verification addresses the last two. Table A.5 shows how the automated techniques described in this report can be used to achieve each of the software quality levels.

The ten levels are in roughly chronological order in terms of when they will be achieved in a test effort. Since different software projects have different standards of performance, the sequence of levels can be used as a criterion for test planning and for when to stop testing. For example, a program that is to be used only with a particular set of inputs needs only to achieve levels one through four. However, if a program must perform a complex sequence of computations for arbitrary sets of inputs, then testing must proceed at least through level eight.

TABLE A.5

TEST TECHNIQUES REQUIRED TO ACHIEVE EACH SOFTWARE QUALITY LEVEL

	Quality Levels									
	1	2	3	4	5	6	7	8	9	10
Code Reading	x									
Static analysis	x	x								
Programmer-defined tests			x							
Structural testing				x						
Functional testing					x	x	x			
Assertion testing								x		
Formal verification									x	x

A.5 GUIDELINES FOR SOFTWARE STANDARDS

Software standards can be divided into two types: (1) those for organizing the software development and (2) those for generating the software. The standards summarized below were used during the development of a large data collection system[1] and are representative of those which can be applied during large programming projects.

A.5.1 Standards for Organizing Software Development

Software development standards are applied to achieve the following goals:

- To give members of the development team access to common information and tools needed in the development process.
- To give the software manager information about the status of the project.
- To make development decisions and validation history available for use in subsequent documentation.

In order to achieve these goals, and to support and monitor the software development effort, a project should include a Software Librarian, a Software Control Notebook, Software Design Specifications, and Software Development Folders.

The Software Librarian maintains information on the status of the project. He must keep the Software Control Notebook and maintain a Problem Report Log. The Librarian serves as a communication point between programmers to minimize interface problems during system

[1] Programming Standards Documents (for the High Energy Laser Data Acquisition and Processing System), Physical Science Laboratory, New Mexico State University, BAAD07-79-C-0192.

integration testing. He also supplies information for preparing progress reports.

A Software Control Notebook should be maintained for each module. It should include the following sections:

- Module name, description, and function
- Local storage and data structures
- File and record format
- Messages to operators and users
- Context of the module in the software system
- Cross reference of global data accessed by the module
- Cross reference of input and output statements in the module

The Software Design Specification should include the following information:

1. Software Subsystem Specifications
   - Identification of modules in the subsystem
   - Calling/called relationship between modules
   - Revision history
   - References to other documentation

2. Module Design Specifications
   - Identification of the module
   - Purpose and function of module
   - Algorithm and strategy of the module
   - Input and output parameter descriptions
   - Peripheral device input and output variables
   - Performance requirements of the module
   - Assumed system state
   - Revision history
   - References to other documents

A Software Development Folder should be created for each module when the module is assigned to a programmer. Its purpose is to ensure the orderly development and modification of the module. The Software Development Folder is the primary management tool used by the Software Librarian to monitor progress during software development. It provides a guide and record of specific programming activities and is used to generate program documentation. The information in the Folder can be used by the Librarian to verify that each module complies with the Design Specifications. The Folder should also document all test results.

A.5.2 Standards for Generating the Software

Standards are imposed on the form of software in an effort to ensure the following:

- Software should be easy to read and understand.
- Software should be self-descriptive and self-checking.
- Software should be easily modified, and all modifications should be traceable.

The following are some examples of software standards:

- The programming language or languages to be used in the development should be specified.
- The software should be designed in a modular fashion. Each module should implement a well-defined function and provide an explicit and clean interface to other modules. Standards for module design have been developed by several sources[1,2].

[1] E. Yourdon and L. Constantine, Structured Design: Fundamentals of a Descipline of Computer Program and Systems Design, Prentice-Hall, 1979.

[2] G. J. Myers, Reliable Software Through Composite Design, Petrocelli/-Charter, New York, 1975.

- The modules should follow a particular style. Each module should begin with comments which describe the purpose or function of the module, list the input and output files used, and define calling parameters and key local variables. They should explain any restrictions on the use of the module, and how abnormal return conditions should be handled. Naming conventions should be established for all symbols in the program.

- Desk checking and code reviews should be performed.

- Static testing should be perfomed using a static analysis tool.

- Modules should be unit tested with instrumentation to collect test case coverage statistics.

A.6 GUIDELINES FOR USING THE TEST TECHNIQUES

The sections below contain information to guide the user in applying each of the test techniques. The recommendations given are general in nature—they should be modified to accommodate the specific needs and problems of each individual software development effort. There may be practical reasons why a test technique cannot be applied. However, in the absence of such obstacles, these guidelines describe how to get the most out of a testing effort and how to test efficiently.

A.6.1 Guidelines for Static Analysis

Static test tools typically have several error detection and program documentation capabilities. Each of these capabilities can usually be invoked separately at the option of the user. Thus, static analysis is a very flexible technique which can serve several purposes. Static testing should be used in the following ways:

- As a programming aid, by providing the programmer with cross-references of variables and other symbols and by indicating errors that are often due to oversight (e.g. uninitialized variables);

- During unit testing, code should be checked for compliance with standards by a static tool. All error and anomaly checking options should be invoked at this stage. In this way, as many errors as possible are removed before more expensive dynamic unit testing or integration testing are performed.

- During integration, static interface checking should be used. This will ensure that the integrated units are compatible in their use of global storage and in the type and number of calling parameters associated with each module.

- When a software product is ready for installation or delivery, static documentation capabilities can be used to provide information for program specifications, reference manuals, and other forms of documentation.

- During the maintenance phase of the program life cycle, static analysis should be used to help to verify the correctness of any modifications to the program. Interface checking should be used whenever a fairly large amount of code is to be added to an existing program.

Since most static analysis options are very inexpensive and can be performed quickly, programmers and testers should be given the freedom to use them whenever they desire. If programmers are permitted to experiment with a static tool, they will quickly learn to feel comfortable with it. Static error checking and documentation production more than pays for itself, since it saves programmers and testers a lot of work that would otherwise have to be done manually.

A.6.2 Guidelines for Assertions

Assertions should be written and included in each module as part of the development process. They should be evaluated during all phases of testing. Assertions should be placed in the following locations in each module, for the reasons noted:

- At the entry and exit points of a module, to check values of incoming and outgoing parameters.
- At input statements, to ensure that meaningful values are accessed.
- Following a call to another module, to be sure that the values returned are acceptable.
- At each control point (branching statement, loop beginning and termination) to check that conditions are compatible with the path taken.
- After complex computations, to prevent propagation of an error.
- To impose conditions that are required to hold over entire sections of code (eg. a loop or module).
- To check histories of computations, which involves comparing the current value of a variable against its previous values.
- To provide safeguards on the integrity of data structures (eg. pointers, counts, value bounds).

A great deal of judgment must be used in determining what assertions to write and how to write them. Thorough testing of a program with assertions alone takes an awful lot of them. During test planning, it should be decided to what degree assertions will be relied upon during testing and whether they will be used to give the finished program fault-tolerance. If a major assertion testing effort is chosen, the specifications and design of the program should be used to develop a complete set of conditions that can be imposed on the code through assertions.

Alternatively, a project manager may decide on a less intensive application of assertions. In this case, the most important conditions required by the design and specifications should be formulated as assertions as the program is coded. Since it is easiest to catch computation errors with assertions, the greatest emphasis when using them to a limited extent should be to check the progress and results of calculations.

Regardless of whether assertions are included in the original plans for testing, programmers should be permitted to use them as debugging aids. Debugging with assertions incurs the additional overhead of preprocessing and compiling the code with its assertions, so some restraints on this method of debugging may be necessary. However, assertions can be more effective and efficient than other techniques, such as execution traces, at detecting and locating errors.

A.6.3 Guidelines for Structural Testing

In Sec. 5.3.1 of this report we list four units of program structure for which coverage may be measured during testing:

- Executable statements
- Branches, which correspond to the outcomes of each decision statement

- Combinations of branches
- Full paths (from entry to exit) of a program

One question that a tester who is considering the use of structural testing will have is which level of coverage should be used. Our answer is that the most important thing to do is to achieve full branch coverage. The reason for this is that it is possible to achieve full branch coverage, and doing so gives the tester the assurance that he has exercised all of the code during testing.

The statement coverage measure should not be used for several reasons: full branch coverage subsumes full statement coverage and is not too much harder to achieve; and testing for branch coverage can provide important information and detect errors that are ignored if only statement coverage is attained.

The combinations of branches and the full paths covered during testing should be identified, but full coverage at these levels is usually impossible. Furthermore, there are other aspects of structural testing that are more important than the sheer numbers of paths or combinations of branches that are covered. During structural testing, the tester should determine whether the flow of control that actually occurs during execution of the program is the same as what was intended during the design phase. The path coverage and execution trace information provided by an instrumentation tool must be used to do this.

Selecting good test data is critical to the success of structural testing. Test cases that are likely to reveal errors should be chosen--candidates for this include singular and extreme values of input variables, and combinations of inputs that represent special conditions that the program must handle. "Meaningless" test cases or test data that is "pulled out of the air" should not be run just to try to drive coverage up—this is inefficient and not likely to result in errors being detected.

Structural testing should begin after an initial level of branch coverage is established during checkout of the code. Structural testing can be combined with other dynamic testing techniques (in particular, assertions and functional testing) by simply measuring the coverage achieved while applying these techniques. However, a set of tests which achieves full branch coverage should be identified as early as possible; this set of test cases should be used to help formulate other tests and for regression testing (retesting after error correction).

For large software systems, it is much easier to obtain complete branch coverage by testing each unit separately. In this way, a significant number of errors can be detected during unit testing that would otherwise not be caught until integration testing was performed[1]. It may not be feasible or desirable to attempt full branch coverage when testing a large system as a whole. However, the experience gained from structural testing at the unit level should be used to form test cases and evaluate test results during system-level testing.

A.6.4 Guidelines for Functional Testing

If formal verification is not performed, the greatest amount of testing effort and resources should be concentrated on functional testing. This is because functional testing amounts to trying to demonstrate that the program works as it is intended for as many cases as possible. Functional testing is open-ended since all possible cases can't be tested. The goal of functional testing is to find and apply a set of test cases that will result in a very reliable piece of software being produced.

[1]M. S. Deutsch, "Software Project Verification and Validation", Computer, April 1981, pp. 54-70.

In Sec. 5.4.1 we discussed the three ways that test data is chosen during functional testing: by examining the functions that the program is to perform, the inputs to the program, and its output behavior. The most important of these, and the first that should be used, is the program functions. In testing the program, its functions should be disaggregated to the smallest level possible. Each identified function should be tested both individually and in combination with others that affect it.

As an illustration, consider a software package which implements dynamic storage allocation and list manipulation through FORTRAN-callable library routines. The package must perform a fairly large number of specific functions, such as: obtaining a block of storage from the operating system, referencing blocks by using pointers, ordering items on lists, etc. Regardless of the method used to design and build the package (top-down, bottom-up, etc.), the code which implements each function should first be tested in as high a degree of isolation as possible. That is, the list-manipulation code should first be tested with data structures artificially created for it; later the storage allocation functions can be used to build the list items.

There are several practices which should be followed to ensure the success of this phase of functional testing. First, it is important to always use relatively small modules that perform one well-defined function. If this is done, the code does not have to be "torn apart" to be tested. Second, functional testing should be included in the unit testing phase rather than put off until system-level testing. Third, a test harness or driver should be used to help isolate and test parts of programs.

The second area of emphasis in functional testing is the inputs to the program. If the program is being written for a limited purpose, and the values of the inputs that will be used whenever the program is run are known, then extensive input testing of the program is not necessary. However, if the program is intended for more general use, then the following things should be tested for:

- The tester should make certain that the program handles all possible input values—even those out of range—gracefully. The program should not count on the user to always provide reasonable values in the proper format.

- The tester should identify "special values" of the input variables which must be handled correctly by the program. Examples of special values include zero for a variable that appears as a denominator, null character strings, numbers with very large and very small magnitudes, etc. These special-value inputs should be tested alone and in combination.

The third area of concern in functional testing is the output behavior of the program. "Stress testing" is the process of searching for ways to produce undesirable behavior in a program. It is usually applied at the system test level, to programs that must react to a wide range of external conditions or that implement approximation schemes or heuristics. Examples of such software include numerical algorithms for approximating complex mathematical functions; garbage collection, list sorting, and other data handling algorithms; and process control systems. Stress testing can be expensive, since typically a large number of test cases must be run. However, it should be used if there is no way to analytically (manually) determine a relationship between program input and output, or when all possible outputs cannot be identified.

A.6.5 Guidelines for Using Formal Techniques

We feel that the formal techniques—symbolic execution and formal verification—are not ready to be put into use in typical software development projects. The techniques are still evolving and the tools that support them are not suitable for use by anyone other than researchers. In its current state, symbolic execution by itself has very limited usefulness as a verification technique, although it shows some promise in supporting other techniques (see Chapter 7).

Our first recommendation to a project manager who is interested in using formal verification on his software would be to consult a researcher in the field about the feasibility of the idea. If the researcher believes that formal verification can be useful in that application, then the project manager might consider hiring him to lead the formal verification effort.

Formal verification must be planned for at the inception of a project, since it affects the requirements and design as well as the code. It is useless to try to apply formal verification to a project that is well along in development—it cannot be added as a "fix" for sections of code that are causing problems. Formal verification can be applied to part of a program only if it is possible to assume that the rest of the program does not affect the properties that are being proved. For example, to verify the security of a data base management system, formal verification must be applied to the routines that read and write the data and to all routines that have "trusted access" to the read and write routines. It is useless to apply formal verification to individual modules in isolation, because then the input assumptions of the module are not examined critically during the proof process.

A.6.6 Knowing When to Stop Testing

There are three reasons to stop a testing effort:

- All resources (time or money) have been used up.
- All tests have been completed.
- The program's performance is satisfactory.

At first glance, being out of resources seems like the worst reason to stop testing. Indeed, it is very undesirable to be unable to complete major portions of a test plan because there is no money or time left for the project. However, the other two reasons given to stop testing usually cannot be used alone. For all but the simplest programming projects, it is impossible to test exhaustively. Unless the details of every use that will be made of the program are known, it is also impossible to know whether the program will always perform satisfactorily. Therefore, all three of the reasons to stop testing must be used together.

These guidelines have described how a test plan should be formed as part of the planning for a programming project. The test plan will consist of tasks which correspond to applying each of the test techniques. These tasks can be separated into those that are fixed in scope, and those that are variable or "open-ended". Our recommendations about when to stop testing can be summarized as: finish all of the fixed-size tasks in the test plan; and perform the open-ended ones to the extent permitted by resource constraints, while making sure that a satisfactory product is produced.

The fixed testing tasks include the following:

- Programming standards checking
- Static error and anomaly checking
- Achieving complete branch coverage
- Checkout of each program function

Each of the fixed-scope tasks has a definite criterion for completeness. Furthermore, if the technique corresponding to each task can be used at all on a program, it can be used to completion. These testing tasks are all inexpensive, and in fact will result in a saving of money and time over traditional testing methods. They should be applied to all programming projects regardless of the reliability requirements of the finished product.

The open-ended testing tasks are:

- Assertion testing
- Full functional testing, including special values testing and stress testing

For programming projects with critical applications, assertions should be used in each of the ways listed in Sec. A.6.2. The tester must strike a balance between developing a very complete set of assertions and the amount of time it takes to write and debug them. Since there are error types that assertions cannot detect, assertion testing should not be allowed to consume all of the resources allocated to the open-ended tasks in the testing effort.

If the formal techniques are not used, functional testing is the last technique to be applied (see Table A.4). A well-planned and managed testing effort should leave a significant amount of resources for functional testing after all of the other testing tasks have been carried out. If this is done, the best policy is to stop testing when all of the remaining resources allocated to testing have been used up. This should allow unanticipated uses of the program to be explored and a high quality product to be produced.

However, if functional testing is stopped because of resource constraints, an effort should be made to determine that the performance of the program will be satisfactory to the end users. One way that this can be done is to compare the error estimate made at the beginning of the test effort (see Sec. A.4.2) with the actual number of errors found during testing. If the number found is much less than the number predicted, the decision to stop testing is suspect.

If formal verification is applied to a software project, the proofs must be carried out to completion in order to conclude that the program has the properties that were subjected to proof. Incomplete applications of formal verification may improve the quality of a program; but we cannot recommend that formal verification be used on a program unless the intent is to complete the effort.

PRECEDING PAGE BLANK-NOT FILMED

APPENDIX B
TEST TOOLS SURVEY

The following pages contain fact sheets on automated software testing tools. These tools have been developed in industry, at universities, and by government organizations either as working test tools or as prototypes based on theories of testing. Each fact sheet presents the following information about a tool:

- Name and originating organization
- Language(s) processed, tool source language, and host computer systems
- A brief description of the tool's purpose and method of operation
- Capabilities and types of analysis performed
- Availability and year of original development
- A key reference for additional information

The capabilities of each tool are given in terms of the testing techniques described in Sec. 2 of this report.

Tools listed as "not commercially available" often may be made available to users either directly by the tool's developers or through contracting government agencies—no attempt was made to determine this information.

The year of origin listed for each tool is intended to give an indication of the length of time that the tool, with most of its current capabilities, has been implemented. Many of the tools are undergoing some development and revision.

The reference given is not necessarily the source of information used in writing the fact sheet. An attempt was made to find a reference in the open literature for each tool that fully described the capabilities of the tool. Most of the tools have user's manuals or other forms of documentation; these were cited as references only where no satisfactory open literature sources were found.

This survey is intended to provide examples of the currently available automated testing tools. It is by no means exhaustive, since a very large number of software tools packages have been developed over the last twenty years. The following sources also contain descriptions of automated test tools:

- Datapro Directory of Software, August 1980, C. 1980 Datapro Research Corporation, Delran, New Jersey 08075.

  A directory of commercially available products, cross-referenced by vendor, tool name, and category. Updated monthly.

- AIAA Software Tool Survey, July 1980. American Institute for Aeronautics and Astronautics Computer Systems Committee, 1290 Avenue of the Americas, New York, NY 10019.

  A compilation of survey forms returned by AIAA members or filled out by Grumman Aerospace Corporation from Datapro and Auerbach. This survey is to be discontinued after July 1980.

- Auerbach Software Reports, Auerbach Publishers Inc., 6560 North Park Dr., Pennsauken, New Jersey 08109.

  Updated monthly.

The fact sheets contained in this survey were compiled solely by General Research Corporation, and were not reviewed by any other tool developers.

We would like to thank Mr. Richard Maitlen of TRW Systems Group for providing information on the TRW test tools.

Index of Test Tools

This index is organized into ten categories of tool capabilities. Within each category the tools are listed in alphabetical order and are followed in parentheses by the languages that they process. Many of the tools appear in several categories. Some of the less-common language dialects are explained in notes at the end of this index.

1. Standards Enforcement

The following tools check programs for compliance with the standard indicated in parentheses.

ANSI FORTRAN Checker and Error Detector (ANSI FORTRAN)

AUDIT (ANSI FORTRAN)

PFORT (PFORT)

TEST COVERAGE ANALYZER/CODE AUDITOR (JOVIAL J73/I)

2. General Static Analysis

The following tools perform one or more of the static analysis checks described in Sec. 2 other than Standards Enforcement.

ACES (FORTRAN)

AMPIC (FORTRAN)

ATDG (FORTRAN)

AUDIT (FORTRAN)

CAVS (COBOL)

DAVE (FORTRAN)

FACES (FORTRAN)

FAVS (FORTRAN)

JOVIAL J73AVS (JOVIAL J73)

PDS (EL1)

PFORT (FORTRAN)

RXVP80™ (FORTRAN)

SADAT (FORTRAN)

SMOTL (SMOD [1])

SQLAB (FORTRAN, PASCAL, JOVIAL J3B-2)

SURVAYOR (FORTRAN)

3. <u>Test Data Generation</u>

The following tools perform test input data generation for programs in the language indicated.

ADAPTIVE TESTER (FORTRAN)

ATTEST (FORTRAN)

SELECT (LISP)

SETAR (none [2])

SMOTL (SMOD [1])

4. <u>Test Harness</u>

The following tools provide assistance in the interactive testing of software. Possible features include automatic test driver generation, evaluation of test input and output data, and assistance with the provision of stub routines.

ADAPTIVE TESTER (FORTRAN)

ATDG (FORTRAN)

AUT (MIL/S)

CAVS (COBOL)

PRUFSTAND (SPL [3])

TESTMANAGER (COBOL)

TPL (FORTRAN)

XPEDITER (COBOL, FORTRAN)

5. Instrumentation

These tools insert probes into the test program source or object code in order to determine execution frequency counts and/or perform timing analysis. All of the tools listed under category 7 ("General Dynamic Analysis") perform instrumentation in addition to other capabilities and are not listed here.

CAVS (COBOL)

FAVS (FORTRAN)

FORTRAN ANALYZER (FORTRAN)

INSTRUMENTERS I & II (FORTRAN)

ISMS (ALGOL 60)

NODAL (FORTRAN)

OPTIMIZER III (COBOL)

PACE (FORTRAN)

PRUFSTAND (SPL [3])

RXVP80™ (FORTRAN)

SADAT (FORTRAN)

TEST COVERAGE ANALYZER/CODE AUDITOR (JOVIAL J73/I)

6. Debugging Aids

These tools provide features which assist in manual debugging of programs. Such features include interactive debugging capability and formatted dumps.

OPTIMIZER III (COBOL)

PRUFSTAND (SPL [3])

TESTMANAGER (COBOL)

XPEDITER (COBOL, FORTRAN)

7. General Dynamic Analysis

These tools provide a combination of the dynamic testing techniques. All provide instrumentation for execution frequency counts.

ACES (FORTRAN)

JAVS (JOVIAL J3)

JOVIAL J73 AVS (JOVIAL J73)

PET (FORTRAN)

SQLAB (FORTRAN, PASCAL, JOVIAL J3B-2)

TAP (FORTRAN)

TPL (FORTRAN)

V-IFTRAN™ (IFTRAN [4])

8. Symbolic Execution

These tools attempt to produce algebraic expressions for the test program's output variables.

AMPIC (FORTRAN)

ATTEST (FORTRAN)

DISSECT (FORTRAN)

EFFIGY (PL/1)

PDS (EL1)

SADAT (FORTRAN)

SELECT (LISP)

SQLAB (FORTRAN, PASCAL, JOVIAL J3B-2)

9. Formal Verification

These tools help to prove mathematically that a program meets its stated specifications.

EFFIGY (PL/I)

PDS (EL1)

PROGRAM VERIFIER (PASCAL)

SELECT (LISP)

SID (GYPSY)

VISTA

10. Mutation Analysis

This technique involves the execution of slightly altered versions of a test program in order to detect errors.

MUTATION ANALYSIS (FORTRAN)

Notes

1 - SMOD is a COBOL-like language developed in the U.S.S.R.

2 - SETAR has not been implemented for a specific language.

3 - SPL is a PL/1-based language.

4 - IFTRAN is a structured FORTRAN superset.

ACES (Automated Code Evaluation System)

Developed by the University of California, Berkeley, for the United States Army Safeguard System Evaluation Agency.

Operates on the special purpose language CENTRAN (similar to FORTRAN). Written in FORTRAN for CDC and IBM machines. Operates in batch mode with user options for tracing variables during the dynamic analysis phase. A program data base which can be used for further analysis is produced during the static analysis phase.

Static Analysis Capabilities:

Detects uninitialized variables.

Coding standards - flags "dangerous" CENTRAN constructs such as the assigned "GOTO".

Flags data flow anomalies.

Identifies unreachable code.

Program documentation - symbol name cross-reference, enumeration of loops.

Reaching set generation.

Dynamic Analysis Capabilities:

Assertions - range checks on variables

Instrumentation based testing - provides path coverage and frequency data.

Not commercially available

Year of origin - 1973

ACES (Automated Code Evaluation System) continued

Reference: C.V. Ramamoorthy, R. E. Meeker and J. Turner, "Design and Construction of an Automated Software Evaluation System," Record - 1973 IEEE Symposium on Computer Software Reliability, New York, pp. 28-37 (April 1973).

ADAPTIVE TESTER

Developed by General Research Corporation, Santa Barbara, California, under contract to the United States Army Ballistic Missile Defense Advanced Technology Center.

A general purpose test harness and performance evaluation package, originally developed to test Ballistic Missile Defense simulations. The tool is written in FORTRAN and can be used to test programs written in any language that produces object code compatible with an overlayed FORTRAN program. The Adaptive Tester can operate in either batch or interactive mode, and consists of four functional parts: (1) a test bed and environment simulator; (2) a performance analysis/data reduction algorithm which operates on the output from a test run; (3) an adaptive algorithm which selects the next set of test data; and (4) a graphical interactive aids package. The nature of the adaptive algorithm for test data selection may be specified by the user; the choices include gradient techniques, user-supplied heuristics, and random number generation. The Adaptive Tester currently resides on CDC 6400/7600 and DEC VAX 11/780.

Dynamic Analysis Capabilities:

Test harness - provides for large numbers of test repetitions (limited only by test program execution time duration), simulation of test program operating environment, interactive supervision of the testing process.

Test data generation - can be performed automatically through gradient, heuristic, or random techniques, or may be supervised interactively by user.

The Adaptive Tester is the property of the U.S. Army. Contact point for use: Mr. Ray Stone, General Research Corporation, P.O. Box 6770, Santa Barbara, CA 93111.

Year of origin - 1976

Reference: D. W. Cooper, "Adaptive Testing," Proceedings - Second International Conference on Software Engineering, San Francisco, (IEEE Catalog No. 76CH1125-4C), pp. 102-105 (October 1976).

AMPIC

Developed by Logicon, Inc., Lexington, Massachusetts.

Operates on FORTRAN code or LITTON assembly. The tool is written in SNOBOL for use on IBM 370. AMPIC builds a representation of a program in terms of certain canonical control structures.

Static Analysis Capabilities:

Module interface parameter type checking.

Flags mixed-mode expressions.

Detects structurally unreachable code.

Documentation - automatic flowcharting, structuring of assembly code.

Symbolic Execution Capabilities:

Program interpretation - determines path conditions.

Algebraic path simplifications for conditions and variables. Detects infeasible paths.

Interactive symbolic execution features - user may specify scope and level of detail to be reported.

Not commercially available

Year of origin - 1975

Reference: M.A. Ikezawa, An Introduction to AMPIC, Logicon, Inc., Report No. CSS-75002, (1975).

ANSI FORTRAN Checker and Error Detector

Developed by Softool Corporation, Goleta, California.

A tool for analyzing FORTRAN programs to determine compliance with the ANSI X3.9-1966 standard for FORTRAN. Ambiguities concerning possible standard violations are handled by "Warning" messages. Versions are available "off-the-shelf" for IBM 360/370 and Data General systems; Softool offers quotations for other systems upon request.

Static Analysis Capabilities:

Coding standards enforcement - checks for compliance with ANSI X3.9-1966 FORTRAN standard. A "general portability" option is also available to check programs for transferrability to other FORTRAN environments.

The permanent license cost for the ANSI FORTRAN checker is $8,000; additional license fee for the portability option is $4,000. Lease plans are available for $360-$480 per month for the Standards checker alone, and an additional $180-240 per month for the Portability Option.

Year of origin - 1977

Reference: Production descriptions are available by calling or writing to:

Softool Corporation
340 South Kellogg Ave.
Goleta, California 93017
(805) 964-0560

ATDG (Automated Test Data Generator)

Developed by TRW Defense and Space Systems Group, under contract to NASA/Johnson Space Center.

A system for interactive testing of FORTRAN programs. The system has three main components: Static Error Analysis (SEA); Unit Test Driver Generator (UTDG); and Path Generation (PATHGEN). The SEA component acts independently; a stand-alone version of SEA is also available. PATHGEN assists the user in generating test input data, and UTDG directs the development of a test driver; once these files are complete, the user can begin test exercises. ATDG supports testing of only one module at a time; however, there is an "external conditions" option which keeps the status of global variables and parameters at entry and exit points of all modules. ATDG is written in FORTRAN and currently resides on a UNIVAC 1110 at Johnson Space Center.

Static Analysis Capabilities:

Program error detection - structurally and logically infinite loops; variables not initialized within a module.

Anomaly detection - local variables set and not used; structurally and logically unreachable code; local variables declared but never referenced.

Program documentation - cross-reference listing of variables and branch predicates.

Dynamic Analysis Capabilities:

Testing facilities - assists user in construction of a test driver.

Test data generation - assists user by indicating path followed for each set of test data.

ATDG (Automated Test Data Generator (continued)

Not commercially available.

Year of origin - 1974

Reference: R.H. Hoffman, and G. L. Houser, <u>User Information for the Interactive Automated Test Data Generator (ATDG) System, Revision 1</u>, NASA/Johnson Space Center, JSC Internal Note No. 75-FM-88, (January 1977).

ATTEST

Developed at the University of Colorado under an NSF grant and subsequently at the University of Massachusetts with support from the U.S. Air Force.

Operates on ANSI FORTRAN code. Requires a source code preprocessor to produce a program variable token list--one version uses DAVE for this purpose. ATTEST operates in batch mode, except that an interactive path selection feature is available. Test paths can be automatically generated in accordance with testing crirteria specified by the user. Once a path has been selected, a symbolic execution algorithm is used to simplify the path predicate conditions, thus producing a set of inequalities. An inequality solving algorithm is then applied to determine input data which will cause the path to be executed.

Symbolic Execution Capabilities:

Test data generation for each selected path.

Program interpretation - path identification, representations of output variables.

Algebraic simplification of path conditions.

Detects unreachable code, as indicated by inconsistent path predicate conditions.

Identifies array subscript violations.

Not commercially available

Year of origin - 1975

Reference: L.A. Clarke, "Automatic Test Data Selection Techniques," <u>Infotech State of the Art Report - Software Testing</u>, Infotech International, Berkshire, England, Vol. 2, pp. 43-63, (1979).

AUDIT

Developed at the Naval Ship Research and Development Center, Bethesda, Maryland.

A standards-enforcement package for testing FORTRAN programs. Compliance with ANSI standards is checked by using a flow-graph analysis of the program under test. The effect of different machine word lengths on program output is evaluated as a means of guaranteeing portability. The original version of AUDIT resides on a CDC 6400.

Static Analysis Capabilities:

Program error detection - detects: structural infinite loops and unreachable code; module interface type and number conflicts; uninitialized variables; recursive calls.

Anomaly detection - enforces ANSI FORTRAN standards (beyond those checked by compiler); flags mixed-mode assignments and expressions; determines "undefinitions" of variables, including EQUIVALENCE pairs.

Dynamic Analysis Capabilities:

Machine word length sensitivity is tested by applying a truncation function to all binary arithmetic operators.

Not commercially available.

Year of origin - 1974

Reference: L.M. Culpepper, "A System for Reliable Engineering Software," IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, pp. 174-178 (June 1975)

AUT (Automated Unit Test)

Developed by IBM.

A test harness system that operates on the object code generated for a program module or modules. A test procedure language MIL-S (Module Interface Language - Specific) is used to control testing and direct the simulation of the test module's operating environment. AUT was one of the first test driver systems available, but has disadvantages in that MIL-S test procedures tend to be lengthy and detailed, and no provision is made for modelling input/output devices or files. AUT operates on IBM 360/370 under DOS or TSO.

Dynamic Testing Capabilities:

Test harness - the procedure language makes possible the recording of tests for use in regression testing. User must supply all test case input data and output specifications for verification.

Available from IBM for rental, $100 per month - 12 month minimum.

Year of origin - 1975

Reference: Automated Unit Test (AUT) Program Description/Operation Manual, IBM Installed User Program Number 5796-PEC, (August 1975).

CAVS (COBOL Automated Verification System)

Under development by General Research Corporation, Santa Barbara, California, under contract to the United States Air Force (Rome Air Development Center).

A general purpose development and testing tool for ANSI COBOL 1968 and 1974 code. CAVS can be operated in batch mode or interactively. Written in a subset of ANSI-COBOL 1974 for use on Univac, Honeywell, and DEC VAX systems.

Static Analysis Capabilities:

Error detection - analyzes code for module interface inconsistencies and uninitialized variables.

Anomaly detection - flags data flow anomalies, unreachable code, improper input/output sequencing.

Documentation - reformatted source listings; cross-references of: calling sequences, file and copy text interactions, identifier set/use, record position set/use, program units.

Dynamic Analysis Capabilities:

Instrumentation based testing - execution frequency data at the program-unit, paragraph, or branch level. The number of input/output operations can also be obtained.

Execution tracing - records the order of execution at instrumented level.

Test history - assists in test case formation, multiple and cumulative test case analysis.

Timing analysis - execution time at the program unit or paragraph level.

CAVS (COBOL Automated Verification System) continued

Not commercially available.

CAVS is currently under development; a working version is scheduled to be installed at RADC in late 1981.

Reference: COBOL Automated Verification System Final Report: Study Phase, General Research Corporation, Report No. CR-3-970, (October 1980).

DAVE

Developed at the University of Colorado under an NSF grant.

Operates on syntactically correct ANSI FORTRAN code; written in FORTRAN. DAVE operates in batch mode--the only user control options are to provide for the handling of non-ANSI constructs. DAVE parses the source code and then performs a depth-first trace of all variables, local and global, in the user program. Data flow across module boundaries via parameter lists and COMMON blocks is included in the analysis.

Static Analysis Capabilities:

Module interface conflicts - type checking, alias detection.

Flags uninitialized variables.

Detects data flow anomalies, including those that occur across module boundaries.

Identifies input and output variables for each module.

In public domain; tape copy available for $100. In use at 35 locations.

Year of origin - 1975

Reference: L.J. Osterweil, and L. D. Fosdick, "DAVE - A Validation Error Detection and Documentation System for Fortran Programs," Software - Practice and Experience, Vol. 6., No. 4, pp. 473-486, (October 1976).

DISSECT

Implemented at the University of California, San Diego. Prior developmental work done at McDonnell Douglas under grant from the National Bureau of Standards.

Performs symbolic evaluation of ANSI FORTRAN programs. Written in LISP. Operates in batch mode. User command options dictate initial values (actual or symbolic) for program variables, specify path to be followed, and indicate what program variables or predicates are to have their values printed at any point in the program.

Symbolic Execution Capabilities:

Algebraic expression simplification of program variables and predicates.

Assertions may be inserted to impose conditions on actual or symbolic values of variables.

Not commercially available.

Version described here completed in 1976 - a predecessor was completed in 1974.

Reference: W.E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Transactions on Software Engineering, Vol. SE-3, No. 4, pp. 266-278, (July 1977).

EFFIGY

Developed at IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

Performs symbolic execution of programs written in a subset of the PL/1 language. EFFIGY itself is written in PL/1 and operates interactively on an IBM/370 under VM/370, using the CMS filing system and context editor. Test programs are restricted to integer-valued variables; variable array indexes are not permitted. User options include tracing variables or statements during execution, inserting "breakpoints" to stop execution at a particular point, and saving the execution state for use during a later test exercise.

Symbolic Execution Capabilities:

Algebraic expression simplification of program variables and predicates.

Proof verification - accomplished by translating verification conditions into path predicates and making consistency checks.

Not commercially available.

Work on EFFIGY was begun in 1973.

Reference: J.C. King, "Symbolic Execution and Program Testing," Communications of the ACM, Vol. 19, No. 7, pp. 385-394, (July 1976).

FACES (FORTRAN Automatic Code Evaluation System)

Developed at the University of California, Berkeley. Research partially supported by the Office of Naval Research.

General purpose static analysis package for ANSI FORTRAN programs. FACES is written in ANSI FORTRAN and operates in batch mode. The tool consists of a source code pre-processor (FFE) which builds a data base, and an analyzer (AIR) which performs correctness checks and provides documentation in response to user requests. Commercial version available for IBM 360.

Static Analysis Capabilities:

Detects structurally infinite loops, invalid nesting.

Flags module interface conflicts - type incompatabilities and aliasing.

Flags uninitialized variables - local variables within a module only.

Enforces certain coding standards if selected by the user.

Produces documentation - cross reference listings by statement, variable, calling sequence, or common block; program graph.

Available for $1590 from COSMIC, University of Georgia, Suite 112 Barrow Hall, Athens, GA 30602.

Year of origin - 1974

Reference: C.V. Ramamoorthy, and S. F. Ho, "Testing Large Software With Automated Software Evaluation Systems," <u>Proceedings - International Conference on Reliable Software</u>, Los Angeles, pp. 382-394, (April 1975).

FAVS (FORTRAN Automated Verification System)

Developed by General Research Corporation, Santa Barbara, California, under contract to the United States Air Force (Rome Air Development Center/ISIE).

A general-purpose static and dynamic analysis tool for programs written in FORTRAN or the structured extension DMATRAN. FAVS will also translate FORTRAN programs into DMATRAN. Operates in batch mode with user options controlling documentation. FAVS is written in DMATRAN and has been installed on CDC 6400, DEC VAX 11/780, HIS 6180, and UNIVAC 1100/80 and 1108.

Static Analysis Capabilities:

Program error detection - single and multiple-module scans for: structurally infinite loops, parameter type and length mismatches, uninitialized variables.

Anomaly detection - flags occurrances of: mixed-mode arithmetic, variables set and not used, structurally unreachable code.

Documentation - symbol cross-reference, common block references, calling sequence listings and matrix

Reaching set generation.

Dynamic Analysis Capabilities:

Instrumentation based testing - branch execution frequency data.

FAVS is owned by the U.S. Air Force. Contact point is Frank LaMonica, RADC/COEE, Griffiss AFB, Rome, New York 13441.

Year of origin - translator was delivered in 1975. Complete capability was delivered in 1978. Resource-efficient version with FORTRAN 77 syntax analyzer was delivered in 1980.

FAVS (FORTRAN Automated Verification System) continued

Reference: D.M. Andrews, and R. A. Melton, Fortran Automated Verification System User's Manual, General Research Corporation, Report No. CR-1-754/1, (April 1980).

FORTRAN ANALYZER

Developed at the Institute for Computer Sciences and Technology of the National Bureau of Standards. The work was partially funded by the National Science Foundation.

Performs statement coverage frequency analysis on ANSI FORTRAN programs. Tool consists of a preprocessor (written in FORTRAN) which inserts calls to a tallying routine at beginning points of branches in user's code.

Dynamic Analysis Capabilities:

Instrumentation based testing - prints frequency of execution of each branch for a single execution of user program.

Not commercially available.

Developed in 1974

Reference: G. Lyon, and R. B. Stillman, "Simple Transforms for Instrumenting FORTRAN Decks," Software - Practice and Experience, Vol. 5, No. 4, pp. 347-358, (October 1975).

INSTRUMENTERS I & II

Developed by Softool Corporation, Goleta, California.

The two INSTRUMENTER tools give execution time and frequency data for FORTRAN programs. INSTRUMENTER I operates at the module level, while INSTRUMENTER II operates at the statement level. Both tools are code preprocessors which insert probes into the source code of the program being tested. Versions are currently available off-the-shelf for IBM 360/370, Data General, and SEL systems; Softool offers quotations for other systems upon request.

Dynamic Analysis Capabilities:

Instrumentation-based testing - execution time and frequency data at the module and statement levels. Data maintained for single or multiple program executions.

The permanent license cost for each of the INSTRUMENTER packages is $5,000; lease plans are available for $225-$300 per month. One year of maintenance is included in the purchase price.

Year of Origin - 1978

Reference: Product descriptions are available by calling or writing to:

Softool Corporation
340 South Kellogg Ave.
Goleta, California 93017
(805) 964-0560

ISMS (Interactive Semantic Modeling System)

Developed at Texas A&M University under a grant from the National Science Foundation.

A system which performs syntactic analysis and execution tracing. The tool has the capability of automatically generating its preprocessor from a target language syntax description written in the language PARSEL. This description is then translated into a preprocessor whose source code is either PASCAL or PL/1. A version of ISMS which tests ALGOL 60 programs has been developed, and a FORTRAN version was in progress as of 1975.

Static Analysis Capabilities:

Program documentation - statement type counts, cross-reference listings.

Dynamic Analysis Capabilities:

Instrumentation based testing - timing and execution frequency data.

Execution tracing - control and data flow tracing with graphic displays; computation tracebacks for selected variables.

Will also provide an estimate of the number of significant figures after each computation.

Not commercially available.

Year of origin - 1975

Reference: R.E. Fairley, "An Experimental Program Testing Facility," IEEE Transactions on Software Engineering, Vol. SE-1, No. 4, pp. 350-357 (December 1975).

JAVS (Jovial Automated Verification System)

Developed by General Research Corporation, Santa Barbara, California, under contract to the United States Air Force (Rome Air Development Center/ISIE).

A general purpose program development and testing package for JOVIAL J3 programs. Operates in batch mode; user controls the tool through a macro command language and directives in the form of special comments placed with the source code. JAVS is written in JOVIAL J3 and FORTRAN, and is currently operational on CDC 6400 and HIS 6080/6180 equipment.

Static Analysis Capabilities:

Program documentation - formatted source listings, calling sequence listings and matrix, cross-reference report.

Reaching set generation.

Dynamic Analysis Capabilities:

Executable assertions - user supplies these to check predicate computations, to maintain bounds on variables, or to trap any special condition in the code.

Instrumentation based testing - branch and module execution frequency data, data on other events according to user designation.

Execution tracing - at instrumented level.
Timing Analysis - execution time by module.

JAVS is owned by the U. S. Air Force. Contact point is Frank LaMonica, RADC/COEE, Griffiss AFB, Rome, New York 13441.

Year of origin - 1975

JAVS (Jovial Automated Verification System) continued

Reference: C. Gannon, "JAVS: A JOVIAL Automated Verification System," Proc. COMPSAC 78 Computer Software and Applications Conference, Chicago, (IEEE Catalog No. 78CH1338-3C), pp. 539-544, (November 1978).

JOVIAL J73 Automated Verification System (J73AVS)

Under development by General Research Corporation, Santa Barbara, California, under contract to the United States Air Force (Rome Air Development Center).

A general purpose development, testing, and documentation tool for the J73 dialect of JOVIAL (adopted as Air Force standard in Spring 1980). The tool can be operated in either interactive or batch mode. It is written in JOVIAL J73 for operation on ITEL AS/5-3 and DEC 20/TOPS 20, and is being developed on a CDC Cyber.

Static Analysis Capabilities:

Error detection - checks for structurally infinite loops, module interface inconsistences, uninitialized variables.

Anomaly detection - flags data flow anomalies; structurally unreachable code; "dangerous" constructs such as ABORT, jumps into CASE or IF constructs, etc.

Documentation - reformatted source listing; symbol cross-reference with set/use; compool description; declarations and references of other JOVIAL J3 constructs; calling sequences.

Reaching set generation - also will display all paths between two points in program.

Dynamic Analysis Capabilities:

Executable assertions.

Instrumentation based testing - execution frequency counts at the program unit, path, branch, or statement level.

Execution tracing - execution sequence information at the program unit, branch, or statement level.

Timing analysis - execution time intervals between any two selected points.

JOVIAL J73 Automated Verification System (continued)

J73AVS is owned by the U.S. Air Force. Contact point is Frank LaMonica, RADC/COEE, Griffiss AFB, Rome, New York 13441.

Year of origin - 1980

Reference: C. Gannon, "A Debugging, Testing, and Documentation Tool for JOVIAL J73," Proc. COMPSAC 80 Computer Software and Applications Conference, Chicago, (October 1980).

MUTATION ANALYSIS

Developed at Yale University under grants from the Office of Naval Research, the Army Research Office, and the National Science Foundation.

A system for detecting errors in ANSI FORTRAN programs by automatically producing programs that are slightly altered versions (mutations) of the program being tested. The rationale for considering mutations is the principle that "experienced programmers write programs which are correct or are almost correct." Mutations are produced by making changes such as the following: changing a constant, replacing a constant by a variable, changing or removing operators, deleting statements, setting predicate values to ".TRUE." or ".FALSE.".

The *strategy* used in testing a program consists of four steps: (1) a set of mutations is constructed; (2) a set of test data is selected and the original program run; (3) each mutant is executed with the test data, and discarded if its output is different from that of the test program; and (4) the remaining mutants are examined for indications of errors in the test program.

Mutation analysis is a dynamic technique, but it can detect errors that static techniques address, such as uninitialized variables, aliasing, data flow anomalies, and unreachable code. Mutation analysis is a sort of "dual" of some test data generation techniques, in that similar error types are detected but the user changes the test data and the tool changes the program.

The Mutation Analysis system operates interactively. Versions exist on a PDP-10 and a CDC 7600. The system is being adapted to handle COBOL and C.

Year of origin - 1977

Reference: T.A. Budd, R. J. Lipton, F. G. Sayward, and R. A. DeMillo, "The Design of a Prototype Mutation System for Program Testing," <u>AFIPS Conference Proceedings - 1978 National Computer Conference</u>, Anaheim, California, pp. 623-627, (June 5-8, 1978).

NODAL (Node Determination and Analysis Program)

Developed by TRW Systems Group, Redondo Beach, California.

Provides statement coverage analysis for FORTRAN programs. Written in machine-independent FORTRAN; versions used on IBM, CDC, GE, and UNIVAC equipment. Operates in batch mode as a preprocessor which passes instrumented code to the compiler. User can save a history file so that coverage statistics can be collected for multiple executions.

Dynamic Analysis Capabilities:

> Instrumentation based testing - prints frequency of execution of each branch of a program; includes multiple executions via history file. Branch execution sequence can optionally be printed with normal program output.

Company proprietary.

Approximate year of origin - 1970

Reference: NODAL is not described in the open literature. TRW has a NODAL User's Manual as documentation support for the tool. Inquiries about NODAL may be directed to:

> Mr. Richard L. Maitlen
> Applied Software Laboratory
> Systems Engineering and Integration Division
> One Space Park
> Redondo Beach, California 90278

OPTIMIZER III

Developed by Capex Corporation, Phoenix, Arizona.

A system of development and debugging aids for COBOL programs running on OS, OS/VS, and DOS/VS systems. The package consists of three parts: OPTIMIZER, which reduces execution time and storage requirements of the object code; DETECTOR, which provides formatted dumps and permits interactive debugging; and ANALYZER, which computes execution timing and frequency data.

Dynamic Analysis Capabilities:

Instrumentation-based testing - execution coverage, frequency, and CPU time data at the statement, paragraph, and module level. Data maintained for single or multiple program executions.

Program debugging - tracebacks of calling sequence and logic in the vicinity of an ABEND; formatted dumps of Working-Storage, Program Registers, Data Division, Memory Map; snap-dumps produced at any point requested; resumption of execution after ABENDs due to Data Exceptions or division by zero.

Program Performance Optimization - execution analysis and timing profiles.

Perpetual license fees from $9,750 to $28,500 depending upon options selected. Leases available for $390 to $1140 per month, depending upon options and duration.

Year of Origin - 1978

Reference: Information regarding Capex produces is available from:

Capex Corporation
P.O. Box 13529
Phoenix, Arizona 85002
(602) 264-7241

PACE (Product Assurance Confidence Evaluator)

Developed by TRW Systems Group, Redondo Beach, California. Some upgrades to and applications of PACE were performed under contract to NASA/MSC.

A collection of automated tools which assist in the planning, production, execution, and evaluation of software projects. The tool documented in the open literature is a statement execution coverage analysis package called FLOW (later TDEM). It is written in FORTRAN for CDC 6400, IBM 360, and UNIVAC 1108. Operates in batch mode on FORTRAN programs; consists of a preprocessor for instrumenting code and two post-processor modules for coverage analysis and tracing.

Dynamic Analysis Capabilities:

Instrumentation based testing - provides statement and module execution frequency counts for single or multiple program executions. For branches not executed, provides listing of statements affecting variables in predicate leading to that branch.

Not commercially available.

Development of PACE began approximately 1971; facilities continue to be added to and upgraded.

Reference: J.R. Brown, A. J. DeSalvio, D. E. Heine, J. G. Purdy, "Automated Software Quality Assurance," Program Test Methods W. C. Hetzel, ed., Prentice-Hall, pp. 181-203, (1973).

PDS (Program Development System)

Developed at Harvard University with support from the Naval Electronic Systems Command.

A package of tools that support interactive program definition, maintenance, and testing. The system operates on programs written in the language EL1. The test tools in PDS include an Integrity Checker which performs some static error checks and a Symbolic Evaluator which can be used for formal verification. PDS also includes facilities for defining, editing, and refining program modules and for incremental verification and retesting of programs.

Static Analysis Capabilities:

Error detection - module interface type conflicts; uninitialized variables (including uninitialized inputs to a module).

Anomaly detection - indicates variables that are set but not used.

Formal Verification Capabilities:

Symbolic Execution - performed using a one-pass analysis, absorbing predicates into conditional expressions. Loops are analyzed by a tool component which tries to find a closed-form expression for the loop variables. (This technique is applied to recursive module calls as well.) The tool develops templates describing the calling parameters, outputs, and "operating environment" of each procedure; the template is referenced when evaluating any procedure call statement.

Verification condition generation - derives these for segments rather than paths, using either user-supplied or incrementally-derived assertions.

Proof generation

PDS (Program Development System) continued

Not commercially available.

Year of origin - 1979

Reference: T.E. Cheatham, J. A. Townley, and G. H. Holloway, "A System for Program Refinement," Proceedings - 4th International Conference on Software Engineering, Munich, pp. 53-62, (September 1979).

PET (Program Evaluator and Tester)

Developed by McDonnell Douglas Astronautics Company, Huntington Beach, California.

Operates in batch mode on FORTRAN programs; written in standard FORTRAN. Currently implemented on CDC 6000/7000 series, IBM 360/370 OS, and UNIVAC 1100. The tool consists of a preprocessor which provides statement instrumentation and documentation and a postprocessor which gives coverage and execution tracing data. PET has an extensive executable assertion capability.

Dynamic Analysis Capabilities:

> Executable assertions - global assertions to check variable ranges, legal or illegal values, array subscript bounds, calling parameter side effects; local (position-specific) assertions formed from any logical expression along with special functions to check array orderings.
>
> Instrumentation based testing - statement, branch, and module execution coverage data; minimum, maximum, first, last values at assignment statements; results of branch predicate evaluations.
>
> Execution tracing - statement sequences in which assertions are violated.
>
> Timing analysis - time spent in each module during execution.

Company proprietary - available for sale for $25,000 from McDonnell-Douglas.

First version originated in 1972; assertions were added in 1975.

References: L.G. Stucki, "New Directions in Automated Tools for Improving Software Quality," Current Trends in Programming Methodology, Volume II: Program Validation, R. T. Yeh, ed., Prentice-Hall, pp. 80-111 (1977).

PFORT (Portable FORTRAN) Verifier

Developed at Bell Laboratories, Murray Hill, New Jersey.

A subset of American National Standard FORTRAN called PFORT has been formally defined by Bell Labs. Programs adhering to the PFORT standard are readily transportable across FORTRAN compilers used by systems throughout the world. The PFORT Verifier is itself written in PFORT and can be used to check compliance of any FORTRAN program with the standard. Programs that comply with PFORT are not necessarily error free, but will produce identical results regardless of the machine on which they are executed. The PFORT Verifier is operational in batch mode on Honeywell 6000, CDC 7600, IBM 360, and UNIVAC 1108 mainframes.

Static Analysis Capabilities:

Flags module interface conflicts - parameter type conflicts, aliasing, common block definition irregularities across modules, correct usage of basic external and intrisic functions.

Coding standards enforcement - enforces all ANSI syntax rules, for instance: no mixed-mode arithmetic, recursive procedure calls, or uninitialized variables used in computations.

Program documentation - symbol and variable cross-reference tables, calling sequences, parameter lists, common block references, global common definitions.

In public domain; used at Bell Labs and Jet Propulsion Laboratory.

Year of origin - 1973

Reference: B.G. Ryder, "The PFORT Verifier," Software - Practice and Experience, Vol. 4, No. 4, pp. 359-377, (October 1974).

PROGRAM VERIFIER

Developed at the University of Southern California Information Sciences Institute, with support from the Department of Defense Advanced Research Projects Agency, and at the University of Texas, with support from the National Science Foundation.

The package is an interactive system for proving PASCAL programs. The system consists of five parts: text editor, parser, verification condition generator, algebraic simplifier, and theorem prover. The system is written in the LISP-based language REDUCE and operates on a PDP-10.

Formal Verification Capabilities:

Algebraic expression simplification - uses manipulation properties of REDUCE to simplify predicates, assertions, and verification conditions.

Verification condition generation - produces and simplifies conditions based on user-supplied assertions and code. Verification conditions are produced separately for each module. The tool assumes that the input and output assertions for other modules hold during this process.

Proof Generation - the tool attempts, under close user interaction and supervision, to establish the validity (or invalidity) of the generated verification conditions. A short time limit is set for the tool to work on its own—if it fails to arrive at a conclusion it prompts the user for more information.

Not commercially available.

Year of origin - 1974

PROGRAM VERIFIER (continued)

Reference: D. I. Good, R. L. London, W. W. Bledsoe, "An Interactive Program Verification System," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, pp. 59-67, (March 1975).

PRUFSTAND

Developed by Software Research Associates, San Francisco, for the Siemers Corporation, Munich, Germany.

A test harness system with facilities for dynamic analysis and interactive debugging. The system was originally designed to test modules written in Assembler and SPL (a PL/1 derivative used for coding systems software). PRUFSTAND operates interactively on the BS2000 timesharing system (similar to IBM OS/VS2 with TSO).

Dynamic Analysis Capabilities:

Executable assertions - compares actual test results with user-specified output values.

Instrumentation based testing - execution coverage and frequency data at the branch level.

Test harness - prompts user for outputs from undefined routines as needed during execution; maintains test case input data and stub interface files: Test driver is compatible with parallel processing.

Program debugging facilities - execution tracing and formatted dumps available at user request; interactive debugging using operating system capabilities.

Year of origin - 1978. Has been used to test components of a large real-time data communications and management system for the German railways.

Reference: H. M. Sneed, and K. Kirchhof, "PRUFSTAND - A Test[illegible] Systems Software Components," Infotech State of the Art [illegible] Software Testing, Infotech International, Berkshire, Eng[illegible] pp. 245-270 (1979).

RXVP80™

Developed by The Software Workshop™, General Research Corporation, Santa Barbara, California.

A general purpose development and testing tool for programs written in FORTRAN or the structured extension V-IFTRAN™. Operates in batch mode, and is compatible with and complementary to V-IFTRAN™. RXVP80™ is written in V-IFTRAN™ and is installable on any computer with at least a 32-bit word length, 50,000 words of storage, and an ANSI X3.9.1966 FORTRAN-compatible compiler.

Static Analysis Capabilities:

Error detection - structurally infinite loops; module interface parameter type and length inconsistencies; uninitialized variables.

Anomaly detection - mixed-mode arithmetic and assignments; data flow (set, not used); structurally unreachable code.

Documentation - formatted source listings; calling sequences; cross-references of common blocks and variable names; input/output statement lists.

Reaching set generation.

Dynamic Analysis Capabilities:

Instrumentation based testing - execution coverage and frequency data at the branch level, for single or multiple test runs.

Available from General Research Corporation for $26,000 (including installation, documentation, training session, warranty).

Released commercially in 1980; has been in internal use since 1972.

Reference: RXVP80™ User's Manual, General Research Corporation, Report No. RM-2333, (1980).

SADAT

Developed at the Kernforschungszentrum Karlsruhe GmbH, Institut fur Datenverarbeitung in der Technik, West Germany.

A general-purpose static and dynamic analysis and symbolic execution tool for FORTRAN. Processes single modules only. Written in PL/1, resides on IBM 370/168. SADAT operates in batch mode.

Static Analysis Capabilities:

Identifies uninitialized variables.

Flags variables that are set and not used.

Detects structurally unreachable code.

Program documentation - provides statement listing by type; symbol usage list; program graph by statement and branch.

Test data generation based upon paths determined from the program graph.

Dynamic Analysis Capabilities:

Instrumentation based testing - produces statement coverage and frequency data.

Symbolic Execution Capabilities:

Algebraic simplification of path predicates, into expressions involving input variables, constants and operators.

The system is available from the developers.

Year of origin - 1978

Reference: U. Voges, L. Gmeiner, A. Amschler von Mayrhauser, "SADAT - An Automated Testing Tool," IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, pp. 286-290, (May 1980).

SELECT

Developed by the Computer Science Group, Stanford Research Institute, Menlo Park, California. Support provided by an NSF grant.

A symbolic execution package which emphasizes path analysis and test data generation. The tool is written in and operates upon programs written in a subset of LISP. SELECT operates in batch mode on a DEC-10. It performs a symbolic execution of the program and simultaneously forms the path conditions for all feasible paths. Test data is then generated for each path by using an inequality solving algorithm. SELECT handles subscripted variables by adding paths; it handles subroutine calls by substituting the code into the calling program. User-supplied assertions can be added to the code to simplify the logic or impose performance requirements.

Symbolic Execution Capabilities:

Program interpretation - produces simplified expressions for path conditions. Detects infeasible paths.

Algebraic expression simplification - for path conditions and program variables.

Test data generation - uses a conjugate gradient algorithm to solve the system of inequalities formed by a path condition. User may affect solution by adding assertions or changing algorithm parameters.

Formal Verification Capabilities:

Proof generation - user supplies verification condition in the form of an output assertion "program" for a path. SELECT determines consistency of output "program" predicates with results of symbolic execution of test program.

SELECT (continued)

Not commercially available.

Year of origin - 1974

Reference: R. S. Boyer, B. Elspas, K. N. Levitt, "SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution," <u>Proceedings - International Conference on Reliable Software</u>, Los Angeles, pp. 234-245, (April 1975).

SETAR

Developed by Logicon, Inc., San Pedro, California.

A test data generation algorithm based on past test cases rather than path selection. The method consists of a two-step iteration: (1) the program is executed with one set of input data, the path taken is identified and its predicate condition determined by symbolic execution; (2) the next test case is chosen so that at least one constraint of all previous path conditions is violated. In step (1) the symbolic execution can proceed with test case values substituted for variables as necessary, since the purpose is to produce any condition that can be violated.

SETAR is currently under development for use on higher level languages. No specific implementation or target language or computer system is specified in the available references.

Dynamic Analysis Capabilities:

Test data generation - feedback from path coverage by the method explained above. User control options include specifying constraints to be violated or imposing constraints to restrict the test domain.

Not commercially available.

SETAR has not been implemented in a stand-alone software package.

Reference: S. Kundu, "SETAR - A New Approach to Test Case Generation," Infotech State of the Art Report - Software Testing, Infotech International, Berkshire, England, Vol. 2, pp. 161-186, (1979).

SID (System for Incrementally Designing and Verifying Programs)

Developed at the University of Texas with support from the National Science Foundation, and at the University of Southern California Information Sciences Institute under funding from DARPA.

An incremental software design, development, and formal verification package. Operates on the language GYPSY; the tool is written in LISP and REDUCE and implemented on a PDP-10. SID has extensive interactive capabilities for design and programming support as well as for testi . Formal verification algorithms may be applied to incomplete programs, modules, or sections of code. Program and specification changes are evaluated without "reproving" unaffected parts of code.

Formal Verification Capabilities:

Verification condition generation - executable code and user-provided specifications and assertion conditions are used to form the path conditions. Conditions are automatically simplified algebraically.

Proof generation - operates interactively in attempting to prove verification conditions. A data base of proof documentation is maintained to assist in forming proofs after modifications are made.

Not commercially available.

Year of origin - 1977

Reference: M. S. Moriconi, A System for Incrementally Designing and Verifying Programs, University of Southern California Information Sciences Institute, Report No. ISI/RR-77-65, (January 1978).

SMOTL

Developed at the Latvian State University, Riga, USSR.

A test data generation system for the COBOL-like language SMOD. The tool attempts to find a set of test data that will provide coverage of all branches in the test program. SMOTL first forms a graphical representation of the test program, performing a few static analysis checks in the process, and then begins building a covering set of paths. The STRATEGY module selects a candidate branch and the ANALYZER module uses symbolic execution techniques to determine the branch's feasibility for the path. Once a covering set of paths has been constructed, it is minimized by combining paths with duplicate branches; then symbolic execution is used again on the resulting paths to determine the test data. SMOTL has been implemented on a Soviet MINSK-32 (160 bytes core, CPU speed about 50,000 op/sec.); SMOTL operates in batch mode.

Static Analysis Capabilities:

Error detection - uninitialized variables.

Anomaly detection - unreachable code.

Test data generation.

Year of origin - 1974. SMOTL has been applied to 39 previously written programs, with satisfactory results for those with less than 300 statements.

Reference: J. Bicevskis, J. Borzovs, U. Straujums, and A. Zarins, "SMOTL - A System to Construct Samples for Data Processing Program Debugging," Infotech State of the Art Report: Software Testing, Infotech International, Berkshire, England, Vol. 2, pp. 13-27, (1979).

SQLAB (Software Quality Laboratory)

Developed by General Research Corporation, Santa Barbara California, under contract with the United States Army.

A verification tool for the following commonly used target languages: FORTRAN and its structured extension IFTRAN; PASCAL and its extension Verifiable PASCAL; and JOVIAL J3B-2. SQLAB perfoms some static checks, and then can be used interactively to perform a symbolic execution. Formal verification of a program may thus be accomplished by the user incorporating assertions into the source code. SQLAB is written in IFTRAN and resides on CDC 6400/7600.

Static Analysis Capabilities:

Error detection - structurally infinite loops; module interface type, mode and number conflicts; uninitialized variables.

Anomaly detection - structurally unreachable code, mixed-mode expressions, unused variables.

Assertion checking - input/output usage declarations, physical units errors.

Dynamic Analysis Capabilities:

Executable assertions - prints violations of user-supplied assertions, perfoms recovery operations in FAIL blocks.

Symbolic Execution Capabilities:

Verification condition generation - uses standard symbolic evaluation and simplification techniques.

Proof generation - consistency with output assertions examined.

Not commercially available.

SQLAP (Software Quality Laboratory) continued

Year of origin - 1977

Reference: S. H. Saib, J. P. Benson, and R. A. Melton, "A Methodology for Program Verification," 1977 Summer Computer Simulation Conference, Chicago, pp. 713-720, (July 1977).

SURVAYOR

Developed by TRW Systems Group, Redondo Beach, California.

Provides static data flow anomaly checks for FORTRAN programs. Operates in batch mode; written in transportable FORTRAN. Performs path analysis for each program module, then determines data flow anomalies along each path.

Static Analysis Capabilities:

Flags uninitialized local variables, including those uninitialized on a particular path.

Data flow anomalies - local and global variables set and not used or set twice without intervening use; flags unneeded common blocks, equivalences, parameters.

Documentation (within program module) - path identification, variable cross-reference.

Company proprietary.

Approximate year of origin - 1973

Reference: SURVAYOR is not described in the open literature. Inquiries about SURVAYOR may be directed to:

Mr. Richard L. Maitlen
Applied Software Laboratory
Systems Engineering and Integration Division
One Space Park
Redondo Beach, California 90278

TAP (Test Coverage and Parameter Evaluation Program)

Developed by General Research Corporation, Santa Barbara, California.

An instrumentation and execution tracing tool for programs written in either FORTRAN or the structured extension IFTRAN. TAP is written in IFTRAN and is operational on IBM 360/370. The tool operates in batch mode in the form of a source code preprocessor and a post-execution data collector.

Dynamic Analysis Capabilities:

Instrumentation based testing - coverage analysis at the branch or statement level.

Execution tracing - initial, final, minimum, maximum values for variables on left side of assignment statements; final value of loop control variable, minimum, maximum, initial, final for loop control parameters; branch counts, final branch taken for IF statements.

Not commercially available.

Year of origin - 1977

Reference: C. Gannon, Testing Coverage and Parameter Evaluation Program: Computer Software System Document, General Research Corporation, November 1978.

TEST COVERAGE ANALYZER/CODE AUDITOR

Developed by Boeing Aerospace Company, Software Quality Engineering, Seattle, Washington.

Provides facility for statement coverage frequency analysis and coding standards enforcement for programs written in JOVIAL J73. The tool is built into the JOVIAL J73/I compiler (the modified version of the JOVIAL J73 compiler obtained from the Air Force in 1978). The code instrumentation takes the form of additional machine-language instructions inserted into the compiled code which provide the execution counts.

Static Analysis Capabilities:

> Coding standards enforcement - the Code Auditor flags constructs that violate specialized standards imposed on the IUS or GSRS projects at Boeing. A listing of these standards is printed as part of the compiler output below the source code listing.

Dynamic Analysis Capabilities:

> Instrumentation based testing - coverage frequency data can be provided at any of three levels: procedure entry points, branch points, or branch points and loop traversals. The frequency data is maintained for one execution of the program only.

In the public domain.

Developed in 1979.

Reference: R. L. Glass, Jovial J73 Software Quality Assurance Tools, Volume I, "Introduction and User Manual," Boeing Aerospace Company, Document No. D180-24975-1, (February 2, 1979).

TESTMANAGER

Developed by MSP (Management Systems and Programming) Inc., Lexington, Massachusetts.

A test harness and debugging system for modularized programs. Written in ANS COBOL for IBM, ICL-S/4; handles several languages. Operates in batch mode.

Dynamic Analysis Capabilities:

Test harness - provides four degrees of complexity of response for environment interface simulation, from simple recording of a call to conditional selection of sets of returns based on previous results. Provides for multiple tests per computer run, file creation and display.

Debugging facility - formatted dumps: gives failure type, contents of parameter areas passed by TESTMANAGER, register/accumulator contents. Output controlled by user options.

Available under perpetual license for $9,000-$13,000 (depending on version, includes maintenance, documentation, training), or 1-5 year rental.

First installed in 1970; currently over 200 users.

Reference: D. Thomas, "Program Testing - Helping Programmers to Help Themselves," Infotech State of the Art Report - Software Testing, Infotech International, Berkshire, England, Vol. 2, pp. 271-281, (1979).

TPL

Developed by General Electric Company, Schenectady, New York.

A test harness system for FORTRAN programs. TPL stands for Test Procedure Language; the language provides a means for controlling the testing process and recording tests for future reference and use. Variable initializations, executable assertions, and execution directions (test start and stop points) are specified in TPL, which has a FORTRAN-like syntax. Stubs for called modules not present in the test code must be provided by the user.

Dynamic Analysis Capabilities:

Executable assertions - can apply locally, over a range of statements, or at test termination. The assertions take the form of FORTRAN logical expressions.

Instrumentation based testing - statement and branch coverage data, expressed as a percentage of statements and branches present in the code being tested, is automatically output.

Test harness - maintains data for multiple test runs, evaluates test success or failure on the basis of assertions provided by the user.

Not commercially available.

Year of origin - 1976

Reference: D. J. Panzl, "Automatic Software Test Drivers," <u>Computer</u>, Vol. 11, No. 4, pp. 44-50 (April 1978).

V-IFTRAN™

Developed by The Software Workshop™, General Research Corporation, Santa Barbara, California.

This tool provides the testing capabilities of executable assertions and code instrumentation within the context of a structured FORTRAN precompiler. V-IFTRAN™ is written in V-IFTRAN™ and can be installed on any computer having a FORTRAN capability.

Dynamic Analysis Capabilities:

Executable assertions - inserted into the code at any point by the user. Violations are brought to the user's attention through use of DEBUG output options. Error recovery can be provided through the use of FAIL block code.

Instrumentation based testing - execution coverage and frequency data at the branch level.

Available from General Research Corporation on 7- or 9-track tape for $6370.

Sold commercially since 1979; has been used internally since 1977.

Reference: V-IFTRAN™ User's Manual, General Research Corporation, Report No. RM-2281 (1979).

VISTA

Developed at the Xerox Research Center, Palo Alto, California.

A formal verification system that can automatically generate assertions needed to form verification conditions. The techniques used include weak interpretation, predicate propagation, assertion generalization, trial assertions, and examination of failed proofs. The starting point of a verification exercise using VISTA is the test program along with user-supplied input and output assertions. Each assertiou generation technique is implemented in a separate module and operates automatically; however, user intervention may be required to determine the next technique to be applied. VISTA uses the theorem prover developed for the PIVOT program verifier.

Formal Verification Capabilities:

Verification condition generation - VISTA works backwards from the output assertion, trying to establish verifiable necessary conditions as assertions for the previous branch.

Proof generation and verification - as trial assertions are generated, they are checked for validity.

VISTA was still under development as of 1975.

Reference: S. M. German, and F. Wegbreit, "A Synthesizer of Inductive Assertions," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, pp. 68-75, (March 1975).

XPEDITER

Developed by Application Development Systems, Inc., San Jose, California.

An interactive testing package for COBOL programs or modules. Interface of the tool with the test program is performed by the Compile Processor, which operates on the compiler output without disturbing either the source or generated object code. Formatted dumps, traces, and selected memory snap-shots are provided by the Dynamic Memory Formatter. Control of the testing process is effected through use of a simple Structural Test Language. XPEDITER supports testing in batch, TSO (online), and IMS environments.

Dynamic Analysis Capabilities:

Test harness - assists with variable initialization and provision of module stubs. Execution may be begun at any point in the program, and may be traced, interrupted, and redirected at user discretion. Multiple test exercises can be performed automatically.

Program debugging facilities - execution traces; formatted dumps; interception of ABENDs; interactive debugging.

Permanent license price: $25,000 for basic system, plus $2,500 for TSO or SPF options and $5,000 for IMS option. Lease arrangements available on request. Maintenance and enhancement charge is 12% of permanent license price annually.

Year of origin - 1980

Reference: Information is available from:

Application Development Systems, Inc.
1530 Meridian Ave.
San Jose, California 95125
(408) 264-2272

APPENDIX C
ANNOTATED BIBLIOGRAPHY

There are several resources in which most important papers on testing can be found. They are repeatedly referenced in this bibliography and, in the case of journals and annual conferences, should be checked in the future for state-of-the-art reports on testing:

Communications of the ACM

Computer (including a special issue on program testing -- Vol. 11, No. 4, April 1978 which contains several tutorial articles, and a special issue on software quality assurance -- Vol. 12, No. 8, August 1979 wich contains several research papers).

IEEE Symposium on Computer Software Reliability, New York City, April 30 - May 2, 1973.

IEEE Transactions on Software Engineering

International Conference on Reliable Software, Los Angeles, April 21-23, 1975.

International Conferences on Software Engineering

Software: Practice and Experience

Software Quality and Assurance Workshop, San Diego, November 15-17, 1978.

In addition, there are several collected bibliographies on testing which were independently prepared:

Hardy, Trotter I., Belkis Leong-Hong, and Dennis W. Fife, Software Tools: A Building Block Approach, National Bureau of Standards, NBS special publication 500-14 (August 1977).

Miller, Edward F. (editor), Infotech State of the Art Report: Software Testing, Infotech International, Maidenhead, Berkshire, England, Vol. 1. pp. 275-305 (1979).

Miller, Edward F. (editor), Tutorial: Automated Tools for Software Engineering, IEEE Catalog No. EHO 150-153, New York (1979).

Riddle, William E., Software Development Environments: A Bibliography, Department of Computer Science, University of Colorado, Boulder, Colorado, Report No. CU-CS-184-80 (June 1980).

The first two are especially important because they are annotated. The last bibliography, which is not annotated, lists more than just testing literature, but contains many newer references not listed in the first two bibliographies.

Rating Scheme

The papers collected in this bibliography have been rated for their contribution to the state-of-the-art of software testing. Each entry is assigned a "star rating" based on the following scale:

**** Superior papers making outstanding contributions to the field,

*** Excellent papers making substantial contributions,

** Good papers making significant contributions, and

* Fair papers making contributions of more limited scope.

The ratings are based on the authors' subjective evaluations of the papers.

C.1 CLASSIC PAPERS (Before 1978)

*** Alberts, David, S., "The Economics of Software Quality Assurance," Proc. of the National Computer Conference, Amer. Federation of Information Processing Societies, New York City, June 7-10, 1976, pp. 433-442.

In this paper Alberts tackles several issues related to the costs of large-scale software systems: cost distributions versus life-cycle phase, the costs of various kinds of errors, the cost reductions provided by different program development and testing methods. He emphasizes that, under current software development preactices, errors are detected later than they should be. This results in very high testing and maintenance costs. The greatest culprits are design errors--he states that in most projects "a dollar more spent in design would have saved five dollars spent on testing and maintenance." He also cites evidence that automated testing techniques can provide a significant cost savings over traditional testing methods.

**** Floyd, Robert W., "Assigning Meanings to Programs," Proc. of Symposium in Applied Mathematics, American Mathematical Society, New York City, April 5-7, 1966.

The fundamental concepts and notations for what has become known as "Floyd-Hoare" programming language semantics were introduced in this paper. Even though the notation is somewhat awkward by current standards, its importance cannot be overstated. Nearly all research in program verification is based on this semantic model.

**** Gerhart, Susan L. and Lawrence Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies," IEEE Trans. on SE, Vol. 2, No. 3, pp. 195-207 (September 1976).

Absolutely <u>must</u> reading for all software engineers, this paper beautifully explains that no formal method exercised by fallible humans will ever guarantee program correctness. Certainly formal methods will increase confidence in correctness, but not even a formal rigorous proof should be completely convincing. To support their claim they cite many cases where authors have presented programs which have been proven correct and had those proofs reviewed by others--yet those programs were incorrect!

Even in mathematics, where the foundations for constructing proofs are much stronger, embarrassing errors in proofs occur. They cite one classic case where a theorem was independently proven in three quite different ways by three mathematicians, all leading to reviewed publications. The theorem was incorrect.

The authors offer some guidance as to how to look for errors in specifications, design, and code, but recognize that these are only guidelines and that there are no guaranteed methods for detecting flaws.

The fact that formal methods do not guarantee correctness should not discourage their use. The authors point out that formalism provides training in rigorous thinking "which is essential for good programming," and provides "an effective language for organizing and expressing knowledge about programs." However, testing must remain a major means to ensure program reliability.

**** Goodenough, John B. and Susan L. Gerhart, "Toward a Theory of Test Data Selection," <u>IEEE Trans on SE</u>, Vol. 1, No. 2, pp. 156-173 (June 1975).

This is probably the most important paper in the testing literature. It has done more than any other work to establish a mathematical framework for the systematic study of software testing. Admittedly the results given are not that profound; indeed, there are really no "deep truths" in the testing field. This does not diminish the significance of their contribution, however.

This paper presents fundamental definitions of "reliability" and "validity" for tesing criteria, and explores the difficulties with attempting to develop criteria with these properties. The authors develop a decision table approach to representing a program and its test data pinpointing the importance of testing each predicate, and where possible, each combination of predicates in a program. They further point out the inadequacies of just structural testing. A program's tests must be generated from the specifications as well as from the program structure if they are to be reliable. They produce guidelines on how to generate test data from specifications.

**** Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," Communications of the ACM, Vol. 12, No. 10, pp. 576-583 (October 1969).

Following up on the ideas introduced earlier by Floyd (1966), Hoare develops rules of inference for simple language constructs.

**** Hoare, C.A.R. and Niklaus Wirth, "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica, Vol. 2, pp. 335-355 (1973).

This paper is a landmark because of its ambitious attempt at axiomatizing a large portion of an actual programming language. Despite a few errors and the fact that not all of Pascal was analyzed, it has become the foundation on which later axiomatizations of Pascal and other languages are based. By showing how well the semantics of a well-designed programming language can be clearly explained, this papeer provided a major push towards developing languages with simple but powerful features.

**** Howden, William E., "Reliability of the Path Analysis Testing Strategy," IEEE Trans. on SE, Vol. 2, No. 3, pp. 208-215 (September 1976).

This is probably the second most important paper ever written in the testing field, second only to Goodenough and Gerhart (1975). There are very few theorems in testing theory because there is so little theory. Many of the most important theorems are presented here. Howden defines a "reliable" test strategy and proves that there are no non-trivial strategies which will be reliable for all programs. He also proves that for each program there is a finite test set which reveals whether the program is correct. He then develops specific definitions for the path analysis testing strategy and proves fundamental theorems about it.

To explore the reliability of path testing for detecting errors, Howden examines several small programs with many errors. He concludes that path testing will detect many errors but will not, in general, detect all errors. For the sample programs examined, 65% of the errors were detected by path testing. He presents theorems which show the conditions under which path testing is reliable for three different error categories. These conditions are fairly strong, so that in general it cannot be assumed they hold. However, even if these conditions are not satisfied, many errors may still be detected.

**** Howden, William E., "Symbolic Testing and the Dissect Symbolic Evaluation System," IEEE Trans. on SE., Vol. 3, No. 4, pp. 266-278 (July 1977).

This classic paper overviews the basic principles of symbolic execution through a discussion of the Dissect system designed and implemented by the author. Dissect symbolically executes ANSI FORTRAN programs. After explaining the features of Dissect, showing how it can detect errors in a simple program, Howden continues with a discussion of the reliability of symbolic execution in general.

Provided the user knows the form of the answer to expect, Howden suggests that symbolic execution will be quite reliable in detecting "path-function" errors; i.e., errors resulting from the wrong computation on the correct path. This is because the resulting function constructed to represent the calculation will "look obviously incorrect." "Path-domain" errors, in which the wrong path is taken, are harder to detect. The function which results may well look reasonable. It is just the wrong funcion for this particular symbolic data; e.g., when a loop iterates one too few times so that a numerical computation is missing a term, but is approximately correct.

Howden recommends that symbolic execution complement rather than replace dynamic analysis. When combined with static analysis tools and tools to analyze software design and specifications for errors, Howden expects the number of errors to be reduced dramatically.

The article is very well-written, with well-chosen examples to illustrate the potential advantages of symbolic execution. The one major flaw in the paper is that because it is necessarily so short, the reader cannot completely grasp how to use Dissect.

*** Huang, J. C., "An Approach to Program Testing," Computing Surveys, Vol. 7, No. 3, pp. 113-128 (September 1975).

This paper overviews how to approach path testing of programs. Its main contribution is the clear "for the masses" style in which it is written. Because it is a survey paper, it does not break new ground; but it does provide an excellent introduction to the problems of path analysis and test data generation.

**** King, James C., A Program Verifier, Ph.D. Dissertation, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, 1969.

King was the first to apply Floyd-Hoare semantics to automatic program verification. In this dissertation he describes a system, operational on an IBM 360, which can automatically verify many simple programs which manipulate integers. His major contribution was in taking an elegant theoretic idea (that of language semantics) and showing how proofs of program correctness could be automatically realized. Of course, as an early prototype system, it was quite limited. Today's program verifiers are much more sophisticated. However, it is a sad reflection on the difficulty of program verification that despite their added sophistication, current systems still cannot automatically verify programs much more complex than those described by King.

**** King, James C., "Symbolic Execution and Program Testing," Communications of the ACM, Vol. 19, No. 7, pp. 385-394 (July 1976).

This is one of the first places in the journal literature where symbolic execution was described. Most earlier papers were either technical reports or conference papers. EFFIGY is described, a system constructed by King to perform symbolic execution on programs written in a simple PL/1 style. Detailed examples illustrating the style of executing software symbolically are given including the user intervention required to direct the symbolic execution.

**** Osterweil, Leon J. and Lloyd D. Fosdick, "DAVE — A Validation Error Detection and Documentation System for FORTRAN Programs," Software: Practice and Experience, Vol. 6, No. 4, pp. 473-486 (October-December 1976).

One of the most frustrating problems when working with most commercial FORTRAN compilers is their poor diagnostic capabilities. For example, most compilers do not perform even modest data flow analysis to check for variables which are referenced without having previously been assigned a value. Several types of data flow anomaly such as this are detected and reported by DAVE, which analyzes ANSI FORTRAN programs. This was one of the first reported systems to perform such analysis. DAVE itself is quite portable because nearly all of it has been written in ANSI FORTRAN. Its utility is enhanced by its ability to handle large programs with many modules. Intermodule data flow analysis detects anomalies across modules. Much of the article details the algorithms employed for this analysis.

**** Ramamoorthy, C. V. and Siu-Bun F. Ho, "Testing Large Software with Automated Software Evaluation Systems," IEEE Trans. on SE, Vol. 1, No. 1, pp. 46-58 (March 1975).

This paper presents an excellent overview of automated software evaluation systems. The importance of tools to lowering lifecycle costs is developed. This is followed by the classification of tools into several categories depending on the nature of the analysis and the point in the lifecycle when the tool is applied. The specific aid which tools can offer, such as checking for loop termination conditions are presented. Many example tools are cited in the bibliography. Some are briefly evaluated in the body of the paper as well.

**** Ramamoorthy, C.V., Siu-Bun F. Ho, and W. T. Chen, "On The Automated Generation of Program Test Data," IEEE Trans on SE, Vol. 2, No. 4, pp. 293-300 (December 1976).

Manual generation of test data is tedious and error-prone. Automated methods are required to reduce the time required to prepare data sets and to help ensure that the data sets have desired properties such as guaranteeing a particular set of paths are taken. In this article the authors discuss automated methods of applying symbolic execution to generate test data. Symbolic execution results in a series of constraints on the input data imposed by the predicates controlling flow of control and perhaps by additional constraints imposed by the user. Provided the constraints on linear or special non-linear forms, known techniques can be used to automatically generate the data. However, there are no algorithmic techniques for arbitrarily complex code.

The authors propose a new method which involves careful selection of values when possible and random selection from a restricted domain when analysis cannot be done. Since random selection can lead to the selection of input values wich will not simultaneously satisfy all constraints, the method includes a backtracking component to undo the selection of input values until a complete set of input satisfying all constraints is created. Of course, since the input domains are effectively infinite for most problems, this approach may not succeed. The authors do not offer any substantiating evidence that the system will work in acceptable time for complex numerical problems and further evaluation is needed before it will be clear whether this system is viable. The algorithm has been implemented in FORTRAN in a system called CASEGEN and is operational on a CDC 6400.

*** Rubey, R. J., J. A. Dana, and P. W. Biche, "Quantitative Aspects of Software Validation," IEEE Transactions on Software Engineering, Vol. SE-1, No. 2 (June 1975), pp. 150-155.

This paper is the first attempt that we are aware of at making a quantitative evaluation of testing techniques. The authors have compiled error data from several software development projects (the sources are kept anonymous), broken down by error type, severity, frequency of occurrence, method of detection, and time of detection. They observe that static analysis and dynamic testing are complementary error detection methods, and that both should be used in an effective software validation effort.

**** Thayer, Thomas A., Myron Lipon, and Eldred C. Nelson, Software Reliability, North-Holland, Amsterdam, 1978.

This is a commercial publication of a final technical report for a study performed by the TRW Defense and Space Systems Group for Rome Air Development Center during 1973-1976. The study analyzes written error reports from five software development projects (sources kept anonymous). Three major topics in the field of software reliability are treated extensively: error classification, causes and prevention of errors, and mathematical modelling of software reliability.

The error classification system appears to be the least satisfactory product of this work. The final system used has 79 error types split among twelve major categories. The error types are not defined other than by a terse one-line description such as "Incorrect operand in equation." Thus there is much room for ambiguity in interpreting the categories and assigning errors to them. The system mixes source code error definitions and run-time symptoms of errors; it also does not distinguish between design and coding errors. However, in the five years since this report has appeared, no better error classification system has been developed.

Chapter 4 presents an extensive analysis of the error data from the five projects. There are regressions of errors versus module size and complexity, time of error detection, and time of introduction of the errors. The effects of programmer abilities and assignments and computer usage on error detection rates are considered. The effectiveness of various design, standards enforcement, and testing tools and techniques are evaluated. Although a large amount of data is considered and an overwhelming number of data reductions are performed, the authors of the study remain unconvinced that they have uncovered error trends that will apply in any general software development setting.

Chapter 5 and 6 are devoted to the mathematical modeling of software reliability. Chapter 5 is a brief summary of several such models; Chapter 6 attempts to apply one model (developed by Nelson) to one of the software projects. These chapters are much more "theoretical" in orientation than the rest of the study—a background and interest in the probabilistic modeling of software is needed to read them in detail. The value of mathematical software reliability theory is not made clear to the more casual reader. The "guidelines to minimize error introduction" which are presented as the conclusions of this analysis seem to be borrowed from structural testing: try not to write unexecutable paths; test all branches in the program, using well-chosen (functional) data; retest all branches affected by a code correction; etc.

This volume is recommended not only for what it offers to the field of software engineering but also as an excellent example of what technical writing should attempt to be. For the most part, the text of this report is delightfully clear and concise. It gives one hope that software engineering may yet be spared from degenerating into a morass of reports documented in pure jargon. Its conclusions and recommendations for improving the reliability of software are somewhat tentative, but that is an accurate reflection of the current state of the discipline. The study provides an excellent foundation for further research.

C.2 RECENT PAPERS (from 1978 to 1980)

*** Andrews, D. M., "Software Fault Tolerance Through Executable Assertions," Twelfth Annual Asilomar Conference on Circuits, Systems, and Computers, Pacific Grove, CA, November 6-8, 1978.

This paper presentes a brief tutorial in how to use executable assertions both to test programs and to provide run-type error recovery. It gives recommendations on where to put assertions to detect processing errors, and describes the kinds of "reasonableness" checks that can be made with them. Coded examples of the use of recovery blocks are presented. A few statistics are presented which show the overhead of using assertions to be acceptable in most applications.

** Bauer, Jonathan A. and Alan B. Finger, "Test Plan Generation Using Formal Grammars," Proc. 4th International Conference on Software Engineering, pp. 425-432, Munich, September 17-19, 1979.

To the extent that a program can be modeled as a finite-state transducer, the large body of theory which has been developed over the last 25 years on transducers can be applied. The authors claim that for certain applications, such as control systems, a finite-state transducer model is appropriate. They have constructed the Automated Test System (ATS) which takes a formal description of the system, maps it into an augmented finite-state transducer, produces a sequence of test cases from that transducer under varying completeness criteria, and runs the tests. This method tends to produce a large number of test cases; however, the ATS can quickly execute each test. The idea is interesting, and is similar to that described by Chow (1978), but the practical impact of their work remains to be seen.

*** Benson, J.P, and S. H. Saib, "A Software Quality Assurance Experiment," Proc. Software Quality and Assurance Workshop, pp. 87-91, San Diego, November 15-17, 1978.

Assertions have been used to augment the weak error-checking capability of many FORTRAN compilers and linking loaders. The utility of executable assertions to detect errors is studied. Errors were seeded into a program of about 1000 lines. Executable assertions were then added to the program. The executable assertions, when violated, cause an error message to be printed. Typically such assertions specify range information such as stating that variable X has a value in the range 0..10, or assertions may state relationships between variables such as stating that the value of variable X is greater than the value of variable Y.

The experiment showed that assertions are quite useful in detecting many common errors. They appear to be most valuable in detecting computational errors, such as using the wrong operator, and weaker in detecting logic and data handling errors. The authors propose further development of new forms of executable assertions to better handle the latter two error categories.

** Bristow, G., C. Drey, B. Edwards, and W. Riddle, "Anomaly Detection in Concurrent Programs," Proc. 4th International Conference on Software Engineering, pp. 265-273, Munich, September 17-19, 1979.

The paper contains a description of algorithms to detect anomalies such as a variable being referenced without ever having been assigned a value. This is a generalization of similar detection capabilities for sequential code. It is essentially a shorter, less developed version of what was later published by Taylor and Osterweil (1980).

*** Budd, Timothy A., Richard J. Lipton, Frederick G. Sayward, and Richard A DeMillo, "The Design of a Prototype Mutation System for Program Testing," Tutorial: Automated Tools for Software Engineering, pp. 226-230, IEEE Catalog No. EHO 150-153, New York, 1979.

The structure and capabilities of a pilot system to perform testing by mutation is described. The system comprises about 10,000 lines of FORTRAN code, and can be used to test FORTRAN programs. Program mutation itself assumes that a program is nearly correct; i.e., differs from the correct version by only a "simple" error such as having a .LT. predicate instead of .LE.. It cannot address radically incorrect problems such as accidentally omitting a whole capability. If a program is incorrect, but a slight "mutation" of it is correct, then the authors argue that a system which automatically mutates programs will be able to detect errors. Their system accepts as input the original program, a user-written description of which classes of mutations to make, and test data on which the original program is known to yield the correct answer. The system will mutate the program according to the description given and run each mutation against the test data. Failure to produce identical answers on the test data "kills" off a mutation. On the other hand, if a mutation yields identical answers, it may be the correct version, and is saved for subsequent analysis, or further testing with new test data.

The authors have found it relatively easy to test subroutines longer than 100 statements, and argue for mutation as a viable test method. This effort is still in its early stages, and the authors recognize much research remains. A major problem is that some mutations may yield programs which are computationally equivalent to the original. Another is in producing large test data sets, although this problem has been addressed elsewhere with some success. The most significant problem facing mutation testing, however, is the astronomical number of potential mutations. In order for this method to become viable, sophisticated methods of reducing the number of mutations of a given program must be found.

***Cheatham, Thomas E. Jr., Glenn H. Holloway, and Judy A. Townley, "Symbolic Evaluation and the Analysis of Programs," <u>IEEE Trans on SE</u>, Vol. 5, No. 4, pp. 402-417 (July 1979).

The capabilities and design of a system for performing symbolic evaluation of programs written in the EL1 language are described. The system was operational on some language features of EL1 at the time the article was written. Symbolic evaluation is a static analysis of a program for the purpose of generating descriptors useful to other tools such as program verifiers and code optimizers. It is actually a collection of analysis techniques for the purpose of producing either closed form formulas or recurrence relations which describe the behavior of each program variable. In this sense it is a generalized symbolic executor.

The work is notable for its ambitiousness: EL1 is a complex language, whereas most earlier work on symbolic execution has been restricted to rather simple language features; loops are analyzed automatically, so that for each variable affected by a loop, the system attempts to construct a closed form formula which describes the input/-output relationship of the loop with respect to that variable; user-defined procedures are analyzed, including any side-effects, an aspect of procedure analysis which is ignored by most symbolic execution systems. The system operates by building a program data-base which captures many logical and structural relationships and then analyzing this data-base. The basic algorithms employed by the system are described in the article.

*** Chow, Tsun S., "Testing Software Design Modeled by Finite-state Machines," IEEE Trans on SE, Vol. 4, No. 3, pp. 178-187 (May 1978).

There are no test strategies which are both reliable and valid for all programs. This fact has led researchers to seek test strategies which are both reliable and valid for a certain class of programs and errors. Chow has developed a strategy for detecting flow of control errors in programs which satisfy certain simplifying assumptions. This method, which he calls "automata theoretic" testing, is both reliable and valid for the special cases for which it was designed. Chow claims that one of the most important advantages of his method is that designs rather than programs can be tested with it. An executable version of the program is not needed--just a design.

First the tester must construct the finite-state machine which characterizes the system's behavior. Chow offers no guidance as to how to do this. He readily admits, however, that only a limited number of applications can be so modeled.

There are three steps in his method: (1) estimate the maximum number of states in the correct design; (2) generate test sequences based on the design; and (3) verify the responses to the test sequences from step 2. The maximum number of states is required so that the tester can be "certain" he has test sequences long enough to force the testing of all possible cycles in the transition diagram. It is not clear how reasonable this assumption is, but it is certainly required in order to guarantee reliability for the method. The test sequences required will force the constructed automaton to go through each transition at least once and also enable indistinct pairs of states to be recognized. The verification step requires a manual walk-through of the design (or automated if executable code is used) and a manual examination of the results for correctness.

There are obvious limitations with this method as the number of states increases. The author cites his personal experience at Bell Labs using this method on three different projects. He was quite pleased with the experience, noting that several errors were detected with this technique.

*** DeMillo, Richard A., R. J. Lipton, and A. J. Perlis, "Social Processes and Proofs of Theorems and Programs," Communications of the ACM, Vol. 22, No. 5 (May 1979), pp. 271-280.

This article openly condemns the study of formal verification as a fruitless pursuit. As would be expected, it prompted a lot of response in the form of letters to the editor and notes in CACM and other software journals. It also provided a challenge to the formal verification community—it is no accident that announcements of successful proofs of significant programs began to appear within a year.

The article argues that proving (verifying) programs lacks an essential element that its counterpart in mathematics—proving theorems--has: the "social process" of peer review of one's results. Proofs of programs are inherently dull, tedious, and uninteresting, so they won't be checked manually. Furthermore, no one will want to perform them manually as a matter of course; and fully automated verification is a long way off and not likely to be helpful in correcting errors. The authors state their conclusions in no uncertain terms: "Even the verifiers themselves sometimes seem to realize the unverifiable nature of most real software;" and "The discontinuous nature of programming sounds the death knell for verification."

Many of the criticisms stated in this article touch on significant problem areas with formal verification, but the general thesis misses the mark. Formal verification is different from mathematics: the former is a quality assurance activity performed by engineers, while the latter is a body of knowledge which is valuable only insofar as it can be proved to be logically consistent and correct. Formal verification of programs can be valuable even if the process is flawed or incomplete--if program proving is found to improve the quality of software in a way that no other software development activity can, then it should be used.

**** Deutsch, Michael S., "Software Project Verification and Validation," Computer, Vol. 14, No. 4 (April 1981), pp. 54-70.

This paper describes the design, development, and testing of a large software development project. The software development and testing proceeded in top-down fashion, with emphasis on identifying each structural element of the program with a specific function. Software development and testing were performed concurrently, as functional capabilities were realized. Automated tools were used to provide documentation, aid in test data generation, instrument the source code to measure test coverage, and analyze and report test results.

Complete branch coverage was required for each module tested. The article includes a very clear explanation of how an instrumentation and test case assistance tool can be used in structural testing. The reports provided by the tools are shown, and each step in the testing process is enumerated and described. Evidence is presented that structural testing used in this manner can save a significant amount of effort (more than 3 person-years) over traditional testing methods.

*** Drasch, Frederick J. and Richard A. Bowen, "IDBUG: A Tool for Program Development," Proc. Software Quality and Assurance Workshop, pp. 106-110, San Diego, November 15-17, 1978.

IDBUG is a tool which automatically constructs a test harness, thereby relieving much of the tedium of dynamic testing. It is implemented in FORTRAN on an HP 21MX-E mini-computer and can generate test modules and monitor their execution. The user specifies the interface between his program and the IDBUG system in a special language. IDBUG is interactive, permitting a programmer to examine data and output, or modify input data as desired.

The authors report that the use of IDBUG on two projects was quite pleasant. They spent much less time on debugging with greater user satisfaction. They felt that the ability to step through different modules during execution gave them an insight into the execution characteristics of the programs they had not previously encountered.

The system as described seems quite useful, and is certainly an improvement over awkward manual methods of constructing testing environments. Its biggest flaw is that the command language is somewhat awkward to use, and could be improved. Such an improvement would be relatively easy compared to the overall development of IDBUG, and would simplify learning and using the tool effectively.

** Duran, Joe W. and John J. Wiorkowski, "Towards Models for Probabilistic Program Correctness," Proc. Software Quality and Assurance Workshop, pp. 39-44, San Diego, November 15-17, 1978.

Since testing cannot, in general, guarantee program correctness, the authors explore the notion of "probability of correctness". Because certain testing methods are more reliable than others, and certain combinations of tests are likely to discover large classes of errors, a probability of correctness can be ascribed to a program depending on the testing it has undergone. The authors study several different testing strategies with respect to random test data and data from special distributions to derive quantitative measures of confidence in the correctness of a program.

One sample strategy for producing a confidence level involved testing a program as a black box on random sample of inputs. This allows them to make statements such as "we have X % confidence that program P has a probability of at least Y of running correctly on an arbitrary input." The more test cases run, the higher will be X and Y. For example, with 1000 test inputs, X = 95% and Y = 0.997.

Such attempts to quantify the degree of correctness of programs is laudable. Such measures as the authors propose appear useful; however, the double probability in their measure makes the significance of the metric much less clear. The other models they propose are much harder to assess since they give so few details. Further work by them is needed to determine how useful these metrics will be in practice, and how easily they can be computed.

** Fitzsimmons, Ann and Tom Love, "A Review and Evaluation of Software Science," Computing Surveys, Vol. 10, No. 1 (March 1978), pp. 3-18.

This paper is long on review and short on evaluation. It is excellent as a summary of the Halstead metrics—it carefully defines each of them, shows how they are computed, and presents interpretations of their meaning. However, the authors seem willing to take at face value the most incredible claims of Halstead and his associates, and the only qualification they issue is that "the data collected to date are not sufficient to verify the hypotheses of software science."

All of the Halstead metrics are based on the operator and operand counts of a program. These values are combined through various formulas loosely based on results in information theory. (There are some ambiguities in the way operators and operands can be classified--the authors do not mention these, however.) The various metrics are supposed to indicate, among other things: the complexity of a program, the level of a language (quantifying the HOL-assembler dichotomy), the effort required to develop a program, the number of errors in a program, and the time required to understand a program.

The authors cite some impressive statistics in support of these properties of the metrics--indeed some are too impressive. The correlation coefficients between predicted and observed program errors are given as .98 and .99 in two experiments in which Halstead himself was involved. These numbers are so high that in the absence of any a priori reason to believe in such a correlation, it seems they must be statistical flukes. The authors are willing to make some very sweeping judgments on the basis of these experiments; for instance, from comparing the effort metric values they conclude that "software projects using PL/I will proceed much faster toward completion than if they used low-level languages. There will be fewer bugs." There are assembler and FORTRAN programmers who wouldn't agree with that at all.

The Halstead metrics are not "magic numbers" that can be used to measure every conceivable aspect of software engineering. Their proponents would do better to stick to trying to approximate more objective quantities such as software costs, errors, and development time, rather than getting involved in emotionally charged issues such as whether one language is superior to another.

** Foster, Kenneth A., "Error Sensitive Test Cases Analysis (ESTCA)," IEEE Trans. on SE, Vol. 6, No. 3, pp. 258-264 (May 1980).

Foster describes a technique he developed for selecting test cases which are likely to find errors. The goal is to find path errors by generalizing on techniques used for testing gates in hardware design. The technique is summarized by three rules for selection. In a simplified form they are: (1) if predicate P depends on (at least) two variables X and Y, then assign values to X and Y in different test cases so that X<Y, X=Y, and X>Y. (2) if X should be less than Y (or greater), then make X differ from Y by the smallest decrement (increment); (3) assign values to each input so its value changes across tests, and so its value differs from that of other variables. The paper elaborates on ad hoc methods to generate such data and argues for the effectiveness of ESTCA based on an analysis of several examples. Foster admits that ESTCA has no theoretical basis, instead it is founded on "pragmatic engineering" considerations. In fact, it closely resembles well-known methods for selecting test data.

** Fujii, Marilyn S., "A Comparison of Software Assurance Methods," Proc. Software Quality and Assurance Workshop, pp. 27-32, San Diego, November 15-17, 1978.

Three software quality assurance methods are compared: quality assurance, acceptance testing, and independent verification and validation. The first ensures sound methods are applied to the overall development process, the second that the customer will find the product acceptable on delivery, and the third helps ensure the product is correct.

*** Gannon, Carolyn, "Error Detection Using Path Testing and Static Analysis," Computer, Vol. 12, No. 8, pp. 26-31 (August 79).

An experiment conducted by General Research Corporation to test the effectiveness of path testing versus static analysis for error detection is described. A 5000-line FORTRAN program was seeded, one error at a time, with 49 errors using the TRW error classification scheme to categorize them. A programmer, working independently from the seeder, was asked to detect and correct all 49 versions of the program using path testing or inspection, whichever seemed more appropriate. To provide a basis for comparison, all 49 errors were also seeded together into the program and run through a static analyzer to determine which errors could have been detected through static analysis alone. The results indicate that neither path testing nor static analysis alone is adequate to uncover all errors. Path testing alone detected about 25% of the errors, mostly logic, computational, and data-base errors. Coupled with inspection, this percentage increased dramatically to 45%, but is still well below an acceptable level. Static analysis, which is most effective in finding data handling, interface, and data definition errors, detected only 16% of the errors. However, one of the errors was not detected by the other methods. Gannon concludes that path testing, static analysis, and inspection should all be part of a comprehensive test plan. She also notes that even these three methods combined are inadequate to detect many errors and that more sophisticated methods are needed.

*** Geiger, Werner, Lothar Gmeiner, Heinz Trauboth, and Udo Voges, "Program Testing Techniques for Nuclear Reactor Protection Systems," <u>Computer</u>, Vol. 12, No. 8, pp. 10-18 (August 1979).

A combination of methods were used to develop and test a nuclear reactor protection system. These methods included static and dynamic testing using SADAT. In addition, multiple versions of the system were independently developed--a technique which proved exceedingly valuable in detecting errors since the independent teams made different types of errors. The software was tested first at the module level, then after integration at the system level. The authors feel the method proved quite effective in uncovering problems in the specifications and in detecting errors in the implementation.

*** Howden, William E., "An Evaluation of the Effectiveness of Symbolic Testing," Software: Practice and Experience, Vol. 8, No. 4, pp. 381-391 (July-August 1978).

Symbolic testing is compared to static and dynamic analysis to determine its reliability in revealing different types of errors. Six independently written programs which contain 28 known errors are the basis for the analysis. The dynamic tests included branch testing, and the static included module interface checking.

The analysis shows that symbolic testing is, by and large, more reliable than any of the other individual methods, such as branch testing. However, when all other methods are combined, symbolic testing is only slightly better. Hence, symbolic execution could be used alone to discover most errors which other standard techniques do, or could be used in conjunction with other methods to ensure greater error detection.

Howden points out that the analysis he conducted was based on just six programs, and that it is not valid to assume that these relationships hold in general. However, even if they are approximately correct, the utility of symbolic execution is supported.

*** Howden, William E., "Algebraic Program Testing," Acta Informatica, Vol. 10, No. 1, pp. 53-66 (1978).

Most testing methods have little mathematical foundation. They are applied because of empirical evidence that they are likely to uncover certain classes of errors. In this paper Howden presents a test method which has a rigorous mathematical foundation. It is reliable for all errors for a certain class of programs. This class is relatively small but important--programs wich manipulate arrays in a specified way, such as sorting an array of elements. The major result of the paper is that a program which satisfies these restrictive conditions is correct if it is correct on a particular set of data. Howden gives a constructive method for producing that data.

The restrictions on the computation within a program to which algebraic testing is applicable are severe enough so that this testing method cannot be widely applied without further research. It is not clear how much further it can be generalized. However, it is a significant contribution to the testing literature because it attempts to support testing strategies with a mathematical analysis of programs rather than ad hoc experience.

** Howden, William E., "DISSECT--A Symbolic Evaluation and Program Testing System," IEEE Trans on SE, Vol. 4, No. 1, pp. 70-73 (January 1978).

This short article adds very little to a longer and much more detailed paper which appeared earlier (Howden, 1977). It explains the purpose and effectiveness of DISSECT, the author's system for symbolic execution. The longer article is a much better examination of DISSECT.

** Howden, William E., "Theoretical and Empirical Studies of Program Testing," IEEE Trans on SE, Vol. 4, No. 4, pp. 293-298 (July 1978).

Two approaches to program testing are described--theoretical and empirical. Theoretical testing relies on a rigorous mathematical foundation which guarantees that if a program and test data have certain characteristics, then the test will uncover a specific class of errors. On the other hand, empirical tests such as path testing rely upon the fact that experience shows that certain types of tests frequently uncover certain classes of errors. In the latter case, there is no guarantee that the method is reliable.

Although theoretical testing is obviously preferred, the known theory of testing is currently so limited that very little actual testing can be accomplished using guaranteed reliable methods. Howden feels that the empirical approach will be the more fruitful for the near future and encourages further research in it.

The paper itself mostly reviews the state of the art very briefly, offering few new insights. It is primarily a position paper by Howden on where he feels research in testing should proceed. Because of his stature in this field, the paper has merit and has probably influenced other workers.

**** Howden, William, "Functional Program Testing," IEEE Trans on SE, Vol. 6, No. 2, pp. 162-169 (March 1980).

"Functional" testing is defined and compared in effectiveness to "structural" testing. Functional testing in its basic form treats a program as a "black-box"; i.e., without examining the program structure and code. Test data is selected from an analysis of the requirements. In its more developed form, functional testing applies the same "black-box" treatment to the design as well, testing each functional component of the design as a black-box. Functional testing has many of the characteristics of "stress" and "boundary" testing; e.g., extreme values in the domain of each input variable should be used as test data.

This short paper is not overly technical and can be read by someone without an extensive background in testing. Howden offers a general philosophy for testing, as well as specifying exact methods in many cases.

He concludes that functional testing is more reliable than structural testing, in that is is likely to find more errors. This is based on a study of a commercial statistical package, IMSL, in which he analyzed all known errors. However, functional and structural testing are complementary, not competing methods, because each will detect errors which the other will not.

** Huang, J. C., "Detection of Data Flow Anomaly Through Program Instrumentation," <u>IEEE Trans on SE</u>, Vol. 5, Nr. 3, pp. 226-236 (May 1979).

A new method of detecting flow anomalies is described and compared to the traditional approach. A flow anomaly occurs when a variable is referenced when its value is undefined, assigned two values without an intervening reference, or assigned a value and then become undefined without an intervening reference. The first case is always an error, the latter two indicate a possible logical error. Classically, flow anomalies are detected statically. Huang proposes instrumenting code to detect them dynamically. This new approach can handle array references, which the static approach cannot. Instrumented programs which undergo structural testing including complete DD-path testing will, as a by-product, detect all data flow anomalies except those dealing with arrays. Arrays require additional testing to cover the entire subscript range.

** Ploedereder, Erhard, "Pragmatic Techniques for Program Analysis and Verification," Proc. 4th International Conference on Software Engineering, pp. 63-72, Munich, September 17-19, 1979.

The Program Development System (PDS) is described. It is a collection of tools used to support program development in the ECL language. The majority of the paper is spend on the central tool--a symbolic evaluator. The system is basically that described in another paper by Cheatham et al. (1979).

** Polak, Wolfgang, "An Exercise in Automatic Program Verification," IEEE Trans on SE, Vol. 5, No. 5, pp. 453-458, (September 1979).

Program verification has been criticized for only being able to handle "toy" programs. The author tries to argue that the state of the art has advanced from "toy" to "small but nontrivial" programs. He supports this claim by showing how a permutation program written by Knuth can be proven correct using the Stanford Pascal verifier[1]. The verifier works from an augmented version of the program in which key assertions are provided by the user.

The paper is not very convincing because the program studied is still a "toy". The original Algol 60 program is just 13 lines. It is slightly longer in the Pascal form which Polak converts it into since he has added lengthy invariant assertions, more comments, and other boilerplate. Furthermore, he has done all of the really hard work by

[1] Stanford Verification Group, Stanford Pascal Verifier, user Manual, Stanford Univ., Dept. of Computer Science, Report No. STAN-C5-79-731, March 1979.

providing the assertions. A more convincing argument for the viability of program verification would have been for the assertions themselves to be automatically generated. A final problem with the permutation program is that it has such a classic set of mathematical properties underlying it. Most programs actually written do not.

*** Reifer, Donald J. and Stephen Trattner, "A Glossary of Software Tools and Techniques," Computer Vol. 10, No. 7, pp. 52-60 (July 1977).

One of the most difficult problems in any field as young and dynamic as computer science is the lack of a standard terminology. The authors provide clear concise definitions for 70 common tool and technique categories such as "compiler" or "test-result processor."

** Scowen, R. S., "A New Technique for Improving the Quality of Computer Programs," Proc. 4th International Conference on Software Engineering, pp. 73-78, Munich, September 17-19, 1979.

If a variable's value is overwritten without ever having been referenced, then presumably there is a logical error in the code. Scowen proposes implementing a hardware check for this condition. The basic flaw with his proposal is that it is not always an error to assign a value without ever referencing it. For example, a value may be computed assuming normal operating conditions, but an exception may cause that computed value to be overwritten. Static analysis could warn the programmer during initial coding that he has a potential error. Scowen's technique, however, could check array references for which subscript bounds cannot be computed statically.

** Sukert, Alan N. and Amrit L. Goel, "Error Modeling Applications in Software Quality Assurance," Software Quality and Assurance Workshop, pp. 33-38, San Diego, November 15-17, 1978.

Different approaches to modeling the prediction of errors are described, including a discussion of the parameters and probability distributions employed. A large project involving 115,000 lines of Jovial code is the basis for the analysis of error prediction models. Detailed error data was available for this project which was tested and became operational in 1973.

All of the models described appear to be somewhat weak, giving answers far different from the actual number of errors found. Unintentionally, the authors seem to point out how poor current models of error prediction really are. They present a plan for using error prediction models as part of an acceptance test. The expected number of errors which will occur over some period of time is computed using an error prediction model. Testing then proceeds over a benchmark period. If the number of errors actually discovered does not exceed the predicted number, the software should be accepted.

*** Taylor, Richard N. and Leon J. Osterweil, "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," <u>IEEE Trans on SE</u>, Vol. 6, No. 3, pp. 265-278 (May 1980).

A data flow anomaly occurs when a variable is referenced without having been previously assigned a value or is assigned a value which is never referenced. The detection of anomalies in sequential code has been studied by many people and several tools exist which detect such anomalies in a variety of languages such as FORTRAN and Pascal. Taylor and Osterweil have extended those results to concurrent code where additional problems in detecting anomalies arise because of the indeterminate order of execution of some statements. They have further isolated several anomalies of a different nature peculiar to concurrent code such as a task never being scheduled, or being scheduled concurrent with itself. The bulk of the paper is taken up describing the algorithms to detect these anomalies. The algorithms are based on a new data structure called a "process augmented flowgraph" which is a graphical representation of a system of communicating concurrent processes. Algorithms found in many optimizing compilers such as those to detect "live" and "dead" variables in code blocks are adopted for this analysis.

** Voges, Udo, Lothar Gmeiner, and Anneliese Amschler von Mayrhauser, "SADAT -- An Automated Testing Tool," IEEE Trans on SE, Vol. 6, No. 3, pp. 286-290 (May 1980).

SADAT is a tool for testing single modules of FORTRAN programs. It is written in PL/1 and runs on an IBM 370/168. SADAT supports static and dynamic analysis plus symbolic execution. Among its capabilities, it can statically detect code anomalies, symbolically execute a particular path in a program, and instrument code to report on characteristics of a particular run such as the DD-paths which were executed.

**** Walker, Bruce J., R. A. Kemmerer, and G. J. Popek, "Specification and Verification of The UCLA Unix Security Kernel," Communications of the ACM, Vol. 23, No. 2 (February 1980), pp. 118-131.

This is one of the first efforts to prove the security of an operating system. Such work represents a change of emphasis in formal verification from attempts to prove the "general" correctness of traditional applications programs. The paper describes the formal specification and verification process in terms that are understandable to a reader who has no previous exposure to the techniques of program proving. It is also an honest account of the difficulties of conducting a proof. Thus, it is highly recommended reading for anyone interested in formal verification.

Much of the work of proving a program is involved in developing a set of formal specifications. These formal specifications must do two things: they must embody the properties that the developers want the software to have; and they must provide a logical basis for the proofs. Three levels of specifications were developed in proving the UCLA Unix security: the top level is a concise statement of what the security criteria is, while the lower levels specify how the software is to implement this criteria. In effect, the software source code provides a fourth level of specification which is imposed on the supporting computer system; however, the proofs stop at the source code level and assume the correctness of the compiler, hardware, etc.

The authors have been encouraged by the success of this effort, but have no illusions about the current practical value of formal verification. They state that "The effort required (in developing the specifications) is sobering. The task is still too difficult and expensive for general use." Likewise, the process of conducting the proofs was made difficult by various factors. Only twenty percent of the code-level proofs were actually completed; yet the total verification effort consumed over four person-years! Part of the reason for this is that the tools and techniques used were constantly undergoing changes. The authors feel that this project at least demonstrated the feasibility of using formal proof methods on operational software.

*** Weyuker, Elaine J. and Thomas J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," IEEE Trans on SE, Vol. 6, No. 3, pp. 236-246 (May 1980).

This important work refines the fundamental effort of Goodenough and Gerhart (1975). Practical problems in constructing criteria that are both "valid" and "reliable" are examined; e.g., a criterion which is reliable and valid on program F may be invalid or unreliable on a slightly modified version of F; i.e., criteria may not be robust.

There are several interesting results. Every criterion for selecting test data is either valid or reliable; i.e., the two notions are not independent! A criterion is "uniformly valid" ("uniformly reliable") if it is valid (reliable) on all programs. If a criterion is both uniformly valid and uniformly reliable then it selects the test consisting of the entire input domain. Hence, no single nontrivial criterion is suitable for selecting test data for all programs.

Since programs are not arbitrarily bad, but are usually almost correct, testing criteria should be able to take advantage of the types of errors typically made. This leads to the notion of a "revealing test criterion." A criterion C is revealing if any one test selected by C exposes a program error then all such tests will. The key to the success of this work lies in the ability to partition the input domain into revealing subdomains for likely types of errors.

*** White, Lee J. and Edward I. Cohen, "A Domain Strategy for Computer Program Testing," IEEE Trans on SE, Vol. 6, No. 3, (May 1980).

The predicates governing the flow of control partition the input domain according to the path taken through a program. A "domain error" occurs when an input leads a program down the wrong path due to an error in the control flow of the program. White and Cohen describe a method for automatically generating test data which will reliabily detect domain errors under specified conditions.

The testing method described has only been studied on simple language features; e.g., arrays and procedures are not allowed. Hence, it currently has limited practical value. However, it is not clear that these features pose insurmountable problems to the method; they simply have not been studied. The method is tractable *if all predicates* in the program are linear. Although this is theoretically quite limiting, the authors present evidence that most non-scientific programs have only linear predicates. If the predicates are linear, the domain is broken into areas bordered by intersecting lines. The method selects test data for each subdomain near its borders, where domain errors are most likely to occur.

** Woodward, Martin R., David Hedley, and Michael A. Hennell, "Experience with Path Analysis and Testing of Programs," IEEE Trans on Metric, SE, Vol. 6, No. 3, pp. 278-286 (May 1980).

Woodward et al. propose a new metric for measuring how effectively a program has been tested. The Test Effectiveness Ratio, or TER, is based on the notion of a "Linear Code Sequence and Jump" (LCSAJ) developed earlier by them. The LCSAJ is based on the program text rather than a flow graph and consists of a sequence of statements executed sequentially and terminated by a statement which causes a non-sequential jump.

The TER is actually a hierarchy of ratios which measure successively more thorough testing. TER1 is the statement coverage ratio, TER2 is the branch coverage ratio, and TER3 is the LCSAJ coverage ratio. TER3 is the inductive base of the hierarchy. Higher levels are attained by testing sequences of LCSAJs; i.e., TERn is the coverage ratio of LCSAJ sequences of length n-2.

After the authors define the hierarchy, they spend the remainder of the paper addressing the fact that long sequences of LCSAJs typically are infeasible and must be "pruned" in order to obtain reasonably high TER values. Hence, their definition of TERn should probably be modified so that only feasible sequences are considered in forming the ratio. Further work must be done to demonstrate the viability of this metric of testing thoroughness.

*** Woodward, M.R., M. A. Hennell, and D. Hedley, "A Measure of Control Flow Complexity in Program Text," IEEE Trans on SE, Vol. 5, No. 1, pp. 45-50, (January 1979).

A static measure of program complexity is proposed and contrasted with McCabe's "cyclomatic number." Unlike McCabe's metric, the number of "knots" in a program is computed directly from a program listing rather than a directed graph constructed from that listing. The listing is augmented by an arrow for each jump which occurs in the program. The arrow connects the source and target lines of the jump. Those places where the arrows intersect are called "knots." Program complexity is said to vary directly with the number of knots.

The advantage of this metric over McCabe's is that it is dependent on the ordering of statements within a program and therefore relates more closely to program readability. There are typically many ways to structure programs which will have the same cyclomatic number, but very different knot numbers. Consequently, the readability of the program, as reflected in statement order, is accounted for in their metric.

This paper argues convincingly that their metric is superior to McCabe's. It is easy to compute and is more intimately tied to the physical layout of the program.

APPENDIX D

AN ASSERTION TESTING EXPERIMENT

Although executable assertions have been discussed in the testing literature for a number of years, no one has attempted to determine what level of resources should be committed to their use in a testing effort. Testing with executable assertions is an open-ended task—it has been left up to the tester to decide what assertions should be used.

This assertion testing experiment sought information which would be helpful to a tester in making this decision. Specifically, we wished to determine:

- Where in a program should assertions be placed?
- How thoroughly can a program be tested using assertions alone?
- What software metrics might provide good indicators of the number of assertions needed for thorough testing of a program?

A set of eight Fortran programs[1] with known errors were chosen for testing. These programs were also used in a previous software testing project,[2] providing a baseline for comparison. The programs are listed at the end of this appendix. They are small, allowing us to thoroughly examine them in the allotted time. The experimental design was:

- Develop a complete set of assertions for the test programs.
- Run the programs and determine the errors detected (Test 1).

[1] B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, McGraw-Hill, 1974.

[2] C. Gannon, R. N. Meeson, and N. B. Brooks, An Experimental Evaluation of Software Testing, General Research Corporation CR-1-854, May 1979.

- Remove the original errors from the programs and add new errors into the corrected versions.
- Using the same set of assertions, test the corrected versions and determine the errors detected (Test 2).
- Examine the relationship between the number of assertions and various measures of program complexity.

D.1 DEVELOPING THE ASSERTIONS

Developing assertions and adding them to existing code ("asserting" the code) was the most time-consuming part of the experiment. Our decisions on where to place assertions were based on both the criteria for candidate locations given in Sec. 7.2 and experience gained during testing.

D.1.1 <u>Easy Assertions</u>

Some program locations were straightforward to assert:

- Control points
- Input statements
- Condition checks on variables with fixed values

An assertion was always placed on a labeled statement at a branch point to assure that the program followed an intended path. If a statement following a branch point was not labeled, an assertion was not necessary since there can be no other way for that statement to be executed.

Some control point assertions seemed unnecessary for these test programs since the programs were small, and all alternative paths could be checked manually. In larger programs, however, this may not be possible. Therefore control point assertions on all labeled statements seem desirable.

Input statements were, for the most part, easy to assert. When we were not familiar with the application area of a program or with meaningful values for input data, we consulted someone more knowledgeable in that field or obtained information from the program's documentation.

As in the case of input statements, checking input and output parameters with assertions is a simple method of describing the assumptions required by a particular module.

By screening input data, one can augment a routine's capabilities; limiting its input and consequently its output. For example, one routine produces real results from real input, but integer results from integer input. A second routine operates on integer data only. By screening input to the first routine, the two routines can be meaningfully combined.

Sometimes it is necessary for a variable to remain constant over an entire section of code. This can be done by introducing a temporary variable to save the value and checking its value using an assertion.

Additional checks that were easily made by assertions were division by zero and overflow in exponentiation. A divisor is simply asserted to be non-zero. Exponentiating to an even power must produce a positive result, while exponentiating to an odd power must produce a result of the same sign as the operand. An assertion violation indicates an overflow.

D.1.2 Hard Assertions

Areas that required more difficult assertions were:

- Computation history checks
- Complex computations
- Condition checks on changing variables

Checking the history of computations by comparing previous results to current results required a thorough knowledge of the program's function. Auxilliary variables are needed to save old values. Formulating assertions of this type was more difficult and time-consuming since it required understanding a program's intent and then deciding where in the code to save and compare values. It is evident that if one is not totally familiar with material being programmed, it is easy to make incorrect assertions. We spent a great deal of time debugging our own assertions. We also found that placing assertions after complex computations was very difficult in programs dealing with unfamiliar subjects. In many cases, assertions were developed by more than one person.

Although making assertion checks for variables that are required to remain constant is fairly simple, determining upper and lower bounds of variables that fluctuate requires understanding a program's intent. A person formulating this type of assertion must be as familiar with the code as the programmer himself.

Other areas that were hard to assert include making equality comparisons with floating point numbers and imposing "reasonableness" conditions on program outputs. Testing for equality between floating point numbers is a poor programming practice, but nevertheless it is used. In these cases, one has to decide what tolerance can be allowed in the comparison and assert that anything else violates correctness of the program. In test program "Floatpt", the squares of any two sides of a triangle are compared to the square of the third side. It was necessary to estimate the tolerance for the numbers representing a right triangle. Thorough knowledge of the use and purpose of a program is essential for this decision since certain applications require different accuracies.

By supplementing programs with assertions that require "reasonable" outputs one can increase error detection. Such assertions can reveal that a program doesn't properly handle certain cases. For example with certain data, the test program "Balance" takes an extra iteration around a loop, recomputes balance and interest and informs a user that "There will be a last payment of 0.00". This erroneous computation was detected by asserting a logical result of greater than half a cent for money balances. This type of assertion cannot be formulated from any specified rule or technique, but rather is a result of insight into a program's purpose.

In summary, two points should be emphasized: (1) we used standards to dictate where assertions should go, and (2) we needed a good understanding of the programs to develop effective assertions.

It was very beneficial to develop general standards for implementing assertions and to adhere to these standards even when the assertions seemed trivial. After developing a technique, we went back over asserted test programs and found many places in the code where we did not include assertions because we considered the code to be perfectly sensible and in no need of checks. Assertions seemed to duplicate code. Assertions would be of little value if correctness of a program could be assured so easily. We found that many errors cannot be observed so simply and adhering to a specific approach is necessary even if the assertions seem superfluous.

One has to understand a program to be able to construct useful assertions. It is inevitable that some assertions will turn out to be incorrect, just as any code initially has bugs in it. These mistakes can be reduced greatly by awareness of a program's function and logic.

A summary of assertion data is given in Fig. D.1. For testing purposes, complex assertions were those which required temporary variables and advanced assertion array constructs. All others were considered simple assertions. The complete programs with simple and complex assertions are listed at the end of this appendix.

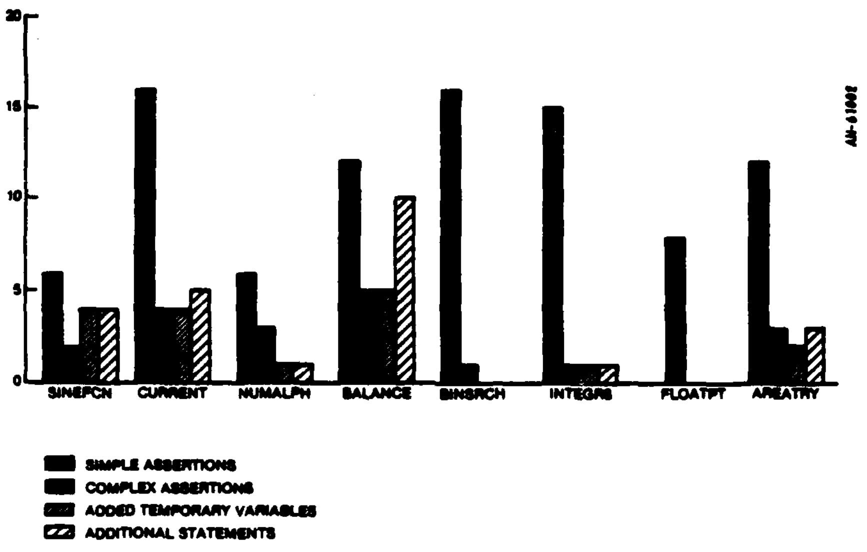

Figure D.1. Assertion Data

D.2 TEST 1 RESULTS

The results of Test 1 are presented in Table D.1. The table also includes the results of the experiment performed by Gannon, et al. (op. cit.) for static and path testing of the same programs. The mixed mode and input type mode errors included in that experiment could not be used in Test 1 because these errors were trapped by the operating system of the machine that we used.

We credited assertions for detecting only errors which directly violated an assertion. One assertion violation can lead to finding multiple errors, but these cases were not considered. (Assertion violations don't pinpoint errors so we had to use judgement in deciding which errors were caught.)

TABLE D.1

TEST 1 RESULTS

ERROR TYPES	SINEFCN	CURRENT	NUMALPH	BALANCE	BINSRCH	INTEGRS	FLOATPT	AREATRY
Unitialized Variable	SA	SA	SA					
Incorrect Computational Logic (2 occurrences)	A PA							
Incorrect Loop Exit	P			PA				
Mixed Mode	S	*+	P+					
Input Type Mode		*+						
Failure to Reinitialize in Loop		PA						
Extra Pass Through Loop				A		PA		A
Incorrect Variable Names (4 occurrences)					SA SA SA SA			
Unchecked Array Bounds					A			
Logical Infinite Loop					PA			
Convergence Logic Error					PA			
Unused Array					S			
Equality Comparison							A	PA
Improper Program Termination				SA			PA	

For each error:

S - found by static analysis[1]

P - found by path testing[1]

A - found in Test 1 of assertion experiment

+ - does not apply to assertion experiment

* - not found by any testing

[1] C. Gannon, R. N. Meeson, N. B. Brooks, An Experimental Evaluation of Software Testing, General Research Corporation CR-1-854, May 1979, p. 2-3.

Errors were present prior to the formulation of assertions in Test 1. Since the errors were known, their location was an important place to include assertions and was useful in developing our standards. The success in detecting errors in Test 1 was partially due to this advantage. Twenty-one out of 24 errors were detected.

Simple and complex assertions did not vary in the errors found except for program "Sinefcn" where one more error was uncovered by complex assertions.

There were many places where more than one assertion was violated by the same error. This leads us to believe that adding many assertions, especially complex assertions, is more time-consuming but uncovers no new errors. However, since assertions do not pinpoint the exact locations of errors, the more assertions violated, the more clues one has in locating errors.

D.3 SEEDING ERRORS INTO THE CORRECTED PROGRAMS

After Test 1 was completed the 24 errors were corrected. In making the corrections, the statement sequences in the original programs were preserved as closely as possible. Only a few changes had to be made in the sets of assertions used to test the original programs after the corrections were made. The corrected programs were run with the same input data used in Test 1 to make sure that the output was correct and that no assertion violations occurred.

Errors were then seeded into the programs. The method of error seeding is described in detail in Gannon, et al.[1] Briefly, the procedure was:

- A static analysis tool was used to tabulate the distribution of statement types in the programs.

[1] C. Gannon, et al., op.cit., pp. 4-1 through 4-24.

- The TRW error categories were examined to determine which ones were appropriate for seeded errors in these programs.

- The frequency of occurrence of errors from each major TRW category was adjusted for the fact that not all major categories were used. Table D.2 gives the percentages of errors in each major category that were seeded.

- A computer program was used to randomly generate a list of candidate sites for the errors.

- For each major TRW error category, the list of candidate sites was manually examined. The following objectives were used to guide the selection of the actual errors.

  - No program was to have a disproportionately large or small number of errors.

  - There was to be no more than one error in any statement.

  - An error could not prevent a program from compiling. The error could result in an abnormal termination if the program produced a reasonable amount of output first.

D.4 TEST 2 RESULTS

The results of Test 2 are presented in Table D.3. The same programs and assertions sampled in Test 1 were used in Test 2. Assertions were put into corrected versions of the programs and later new errors were seeded. This eliminated any advantages due to relations between errors and assertions as present in Test 1. As expected, error detection in Test 2 was not as successful in Test 1. Twenty-five out of the 34 seeded errors were detected.

TABLE D.2
ERRORS FROM MAJOR TRW ERROR CATEGORIES SEEDED IN TEST 2

Error Category	Percentage in TRW Project 5 Study*	No. Used in Test 2	Percentage Used in Test 2
A_000 Computational	12.7%	6	17.6%
B_000 Logic	24.5%	11	32.4%
C_000 Data Input	3.9%	2	5.9%
D_000 Data Handling	11.0%	5	14.7%
G_000 Data Definition	8.9%	4	11.8%
H_000 Data Base	12.5%	6	17.6%
Totals	72.9%	34	100.0%

*Data derived from Table 4.2, page 4-16 of the TRW report.

Figure D.2 is a summary of Test 1 and Test 2 assertion data. Complex assertions had more bearing on error detection in Test 2—they detected five errors that simple assertions did not. It can be seen from this that complex assertion violations not only aid in locating errors which have already violated simple assertions but also uncover new errors that may otherwise be overlooked.

TABLE D.3

TEST 2 RESULTS

ERROR TYPES	SINEFCN	CURRENT	NUMALPH	BALANCE	BINSRCH	DIVERGE	FLOWPT	ANALYZE
A_300 Sign Convention Error						CMX SPL	CMX SPL	
A_600 Incorrect Equation Used		CMX SPL						
A_100 Incorrect Operand in Equation		*				CMX SPL		
A-500 Computation Produces Over/Underflow						CMX SPL		
B_400 Missing Logic or Condition Tests	CMX		* | CMX	CMX SPL				
B_700 Duplicate Logic					CMX SPL			
B_300 Wrong Variable Being Checked				*				CMX SPL
B_200 Logic Activities Out of Sequence				CMX SPL | CMX SPL				
B_100 Incorrect Operand in Logical Expression							*	
B_500 Too Many/Few Statements in Loop		CMX SPL						
C_100 Invalid Input Read From Correct File					CMX SPL			CMX SPL
D_100 Data Initialization Not Done		*		CMX				
D_400 Variable Ref. by Wrong Name			CMX					
D_500 Incorrect Variable Type	CMX SPL							
D_600 Subscripting Error					CMX SPL			
G_100 Data Not Properly Defined			*		CMX			
G_300 Data Being Ref. at Incorrect Loc.			CMX		*			
E_200 Data Init. to Wrong Value	*					CMX SPL		CMX SPL
E_100 Data Not Init. in Data Base							CMX SPL	CMX SPL
E_300 Data Units Incorrect		*						

For each Error:

CMX - found by a complex assertion

SPL - found by a simple assertion

* - not found

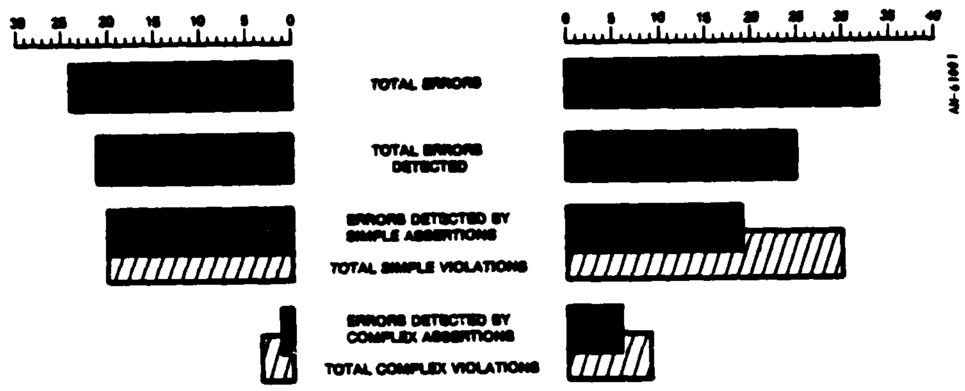

Figure D.2. Summary of Test 1 and Test 2 Results

D.5 ASSERTIONS AND COMPLEXITY MEASURES

The test programs' complexities were evaluated using Halstead, McCabe, and Magel metrics. Halstead measures[1] are based on program length and volume. These measures are derived from the number of distinct operators and operands and their frequency of use in the program. McCabe measures[2] are based on program control flow. The properties of programs used for computing McCabe metrics are the number of control points and branches. Magel metrics[3] are based on program

[1] A. Fitzsimmons and T. Love, "A Review and Evaluation of Software Science," ACM Computing Surveys, Vol. 10, No. 1, March 1978, 3-18.

[2] T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, 308-320.

[3] K. Magel, "Regular Expressions in a Program Complexity Metric," SIGPLAN NOTICES, Vol. 16, No. 7, July 1981, 61-65.

control flow "regular expressions" which show all possible execution sequences of a program. Magel's metric is computed from the total number of operands, operators, and parentheses in the minimally parenthesized regular expression.

Figure D.3 lists the values of the metrics and other information pertaining to complexity measures. The two test programs that did not conform to our expected relation between complexity and number of assertions were Integr8 and Floatpt.

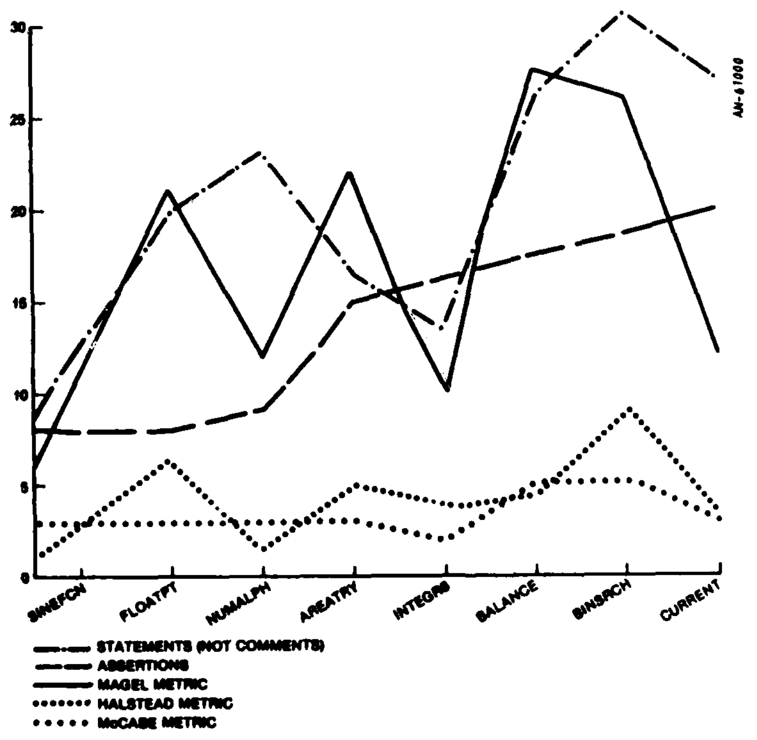

Figure D.3. Complexity Measures

Program "Integr8" rated lowest in McCabe metric, low in Halstead metric, but contained quite a few assertions. This is because this program does not have any input variables. This situation is uncommon. Since the program works on only one set of data, more assertions could be written to provide tight bounds on the acceptable values of variables.

Program "Floatpt" rated high in Halstead metric, intermediate in McCabe metric, and had few assertions. There are several reasons for this discrepancy. "Floatpt" does not have a loop, thereby eliminating the need for assertions that check conditions over loop iterations. "Floatpt" is the only program which includes error-handling provisions for improper input, so there were no input check assertions. Although its algorithm can be stated in one English sentence (the Pythagorean Theorem), expressing this in a standard programming language is difficult and requires three distinct cases. This increases the number of operands and operators resulting in the high Halstead rating.

In summary, it appears that the number and complexity of assertions required for an effective test are based on a different combination of program properties than either the Halstead or McCabe metrics. From the test cases analyzed, it can be seen that complexity measures useful for estimating the number of assertions required by a program should be based on the following:

- The number of loops
- The number of control points
- The number of complex computations[1]
- The number of the input data values expected

[1]We counted complex computations by looking at the number of arithmetic or logical expressions that had two operators. Very complicated expressions could have several complex computations embedded in them; for example, we counted 5 "complex computations" in the statement

```
AI = E/SQRT(R**2+(6.2832*F*L-1.0/(6.2832*F*C))**2)
```

since it has 11 arithmetic operators.

These properties and their relation to each program are shown in Fig. D.4. Loop conditions seem to require the most complex assertions. Control points require the largest number of assertions. Computation and input checks require an assertion for each variable or operation involved.

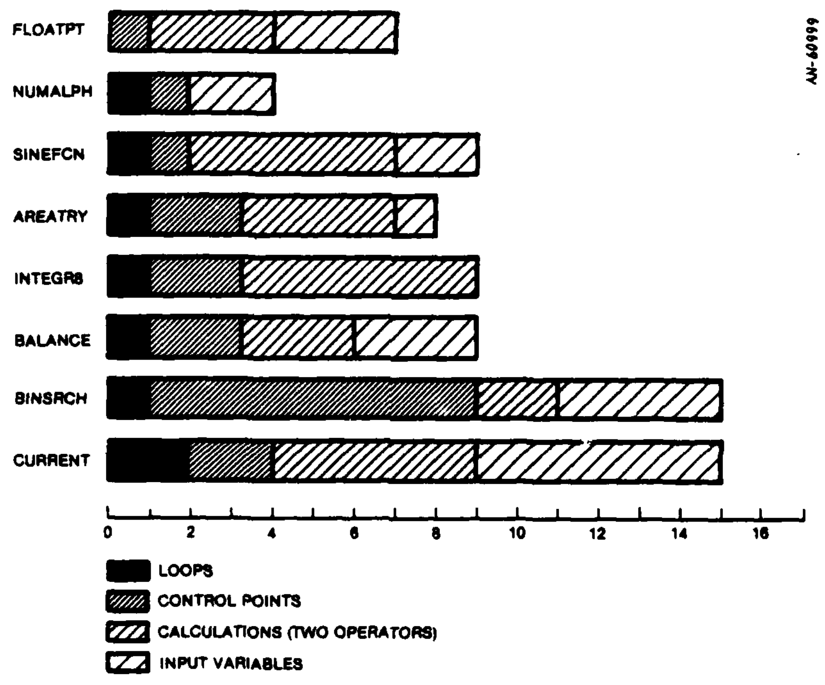

Figure D.4. Program Properties Significant to Assertion Development

D.6 CONCLUSION

A unique set of assertions is necessary for each program, which requires a great deal of time and effort. There are other testing methods that can operate generally on any program without changes or additions. Simpler methods of software testing, such as static analysis, can recognize many errors but the tested programs may still contain flaws. Ultimately, assertion testing is more demanding but detects more subtle errors. Its usefulness in testing may depend on strictness of requirements and how crucial certain software is to a system.

Few studies have been done on the effectiveness of assertion testing and there is a need for further experimentation. Tests should be performed on more extensive programs in various programming languages.

LISTINGS OF THE PROGRAMS USED IN THE ASSERTION TESTING EXPERIMENT

```
      PROGRAM AREATRY
C.ASSERT=ON
C
C     FIRST ATTEMPT FOR APPROXIMATING AREA UNDER A CURVE
C
C     SOURCE =    KERNIGHAN AND PLAUGER
C                 THE ELEMENTS OF PROGRAMMING STYLE
C                 PAGE 120.
C
    1 AREA = 0.0
      READ (5,10)T
      ASSERT (T .GT. 0.)
   10 FORMAT (F10.4)
      H = 0.1
      HINC = H
      ASSERT (H .LT. T)
      X = 0.0
      TEMP = 0.
    2 XN = -X
      ASSERT (T .GT. X)
      TEMP = TEMP + H
      ASSERT (TEMP .LE. T)
      ASSERT (TEMP * H .LE. T)
      ASSERT (XN .LE. 0.)
      AREA = AREA + (6.0 * (2.0**XN) + 6.0 * (2.0**(XN-H))) * 0.1 / 2.0
      ASSERT (AREA .GT. 0.)
      ASSERT (AREA .LE. X*6.0 * (2.**0.))
      ASSERT (H .EQ. HINC)
      X = X + H
      ASSERT (X .GT. 0.)
      ASSERT (X .LE. T)
      IF (X - T) 2,8,9
    8 WRITE (6,33) AREA
      ASSERT (X .EQ. T)
      ASSERT (AREA .GT. 0.)
      ASSERT (AREA .LE. 6.0 * T)
   33 FORMAT (7H AREA =,F8.5)
      GOTO 1
    9 CONTINUE
      ASSERT (X .GT. T)
      CALL EXIT
      END
      PROGRAM BALANCE
C.ASSERT=ON
C
C     COMPUTES A TABLE OF MONTHLY BALANCES AND INTEREST CHARGES FOR
C     A GIVEN PRINCIPAL AMOUNT, INTEREST RATE, AND MONTHLY PAYMENT.
C
C     SOURCE = KERNIGHAN AND PLAUGER
C              THE ELEMENTS OF PROGRAMMING STYLE
C              PAGE 107.
C
C     CONVERTED TO FORTRAN         7/11/78        REG MEESON
C
      REAL  A,R,M,B,C,P
C
   10 READ (5,101) A,R,M
      ASSERT (A .GE. 0. .AND. R .GE. 0. .AND. M .GE. 0.)
      ASSERT (M .LE. A + A*R/1200.)
  101 FORMAT (3F10.4)
      WRITE (6,102) A,R,M
  102 FORMAT (14H THE AMOUNT IS,F10.2,
     $         23H   THE INTEREST RATE IS,F6.2,
     $         25H   THE MONTHLY PAYMENT IS,F8.2)
      IF (M .LE. A*R/1200.) GOTO 30
      WRITE (6,103)
  103 FORMAT (1H-,
     $'        MONTH        BALANCE    CHARGE     PAID ON PRINCIPAL' /)
      TEMPI = 0.
      TEMPRIN = 0.
      TEMPINT = 0.
      RATE = R
      B = A
      TEMPBAL = B
      DO 18 I = 1,60
      C = B*R/1200.
      ASSERT (R .EQ. RATE)
      TEMPI = TEMPI + C
      ASSERT (I .EQ. 1 .OR. C .LE. TEMPINT)
      TEMPINT = C
      ASSERT (C. LE. M)
      IF (B+C .LT. M) GOTO 20
      ASSERT (B+C .GT. M+.005)
      P = M - C
      ASSERT (P .GE. 0.)
      ASSERT (P .LE. M)
      ASSERT (P .LE. B)
      ASSERT (P .GE. TEMPRIN)
      B = B - P
      ASSERT (B .LE. A)
      ASSERT (B .LE. TEMPBAL)
      TEMPBAL = B
     TEMPRIN = P
     ISAVE = I
18   WRITE (6,181) I,B,C,P
181  FORMAT (I13,3F13.2)
20   BPLUSC = B + C
     ASSERT (BPLUSC .GT. 0.005)
     ASSERT (C .EQ. R*B/1200.)
     WRITE (6,201) BPLUSC
201  FORMAT ('OTHERE WILL BE A LAST PAYMENT OF    ',F8.2)
     ASSERT (M*ISAVE + BPLUSC .GT. A+TEMPI-.005)
     GOTO 10
30   WRITE (6,301)
     ASSERT (M .LT. A*R/1200.)
301  FORMAT ('OUNACCEPTABLE MONTHLY PAYMENT')
     GOTO 10
     END
```

```
      PROGRAM BINSRCH
C.ASSERT=ON
C
C     BINARY SEARCH PROCEDURE TO FIND AN ELEMENT *A* IN A TABLE *X*
C     THE ELEMENTS IN *X* MUST ALREADY BE SORTED INTO INCREASING ORDER
C
C     SOURCE = KERNIGHAN AND PLAUGER
C              THE ELEMENTS OF PROGRAMMING STYLE
C              PAGE 110.
C
      DIMENSION X(200),Y(200)
    1 READ (5,50,END=999) N
      ASSERT (N .LE. 200)
      ASSERT (N .GE. 1)
   50 FORMAT (I5)
      READ (5,51) (X(K),Y(K),K = 1,N)
      ASSERT (.ALL. K .IN. (2,N) (Y(K) .GT. Y(K-1)))
   51 FORMAT (2F10.5)
      READ (5,52) A
   52 FORMAT (F10.5)
      IF (X(1) - A)41,41,11
   41 IF (A - X(N))5,5,11
   11 CONTINUE
      ASSERT (A .LT. X(1) .OR. A .GT. X(N))
      PRINT 53,A
   53 FORMAT(1H ,F10.5,
     1     26H IS NOT IN RANGE OF TABLE.)
      GOTO 1
    5 LOW = 1
      ASSERT (X(1) .LE. A)
      ASSERT (A .LE. X(N))
      IHIGH = N
    6 IF (IHIGH - LOW - 1)7,12,7
   12 CONTINUE
      ASSERT (XLOW .LE. A)
      ASSERT (A .LE. XHIGH)
      ASSERT ( IHIGH .EQ. LOW+1)
      PRINT 54,XLOW,YLOW,A,XHIGH,YHIGH
   54 FORMAT (1H 5F10.5)
      GOTO 1
    7 CONTINUE
      ASSERT (LOW .LT. IHIGH-1)
      MID = (LOW + IHIGH)/2
      ASSERT (LOW .NE. MID .OR. MID .NE. IHIGH)
      IF (A - X(MID))9,9,10
    9 IHIGH = MID
      ASSERT (IHIGH .GE. LOW)
      ASSERT (IHIGH .LT. N)
      ASSERT (A .LE. X(MID))
      GOTO 6
```

```
 10 LOW = MID
    ASSERT (LOW .LE. IHIGH)
    ASSERT (LOW .GT. 1)
    ASSERT (A .GT. X(MID))
    GOTO 6
999 STOP
    END
```

```
      PROGRAM CURRENT
C.ASSERT=ON
C
C     CURRENT COMPUTING PROGRAM
C
C     SOURCE = KERNIGHAN AND PLAUGER
C              THE ELEMENTS OF PROGRAMMING STYLE
C              PAGE 103.
      REAL  L
C
C     INPUT VALUES FOR RESISTANCE, FREQUENCY AND INDUCTANCE
      READ (5,20) R,F,L
      ASSERT (R .GE. 0. .AND. F .GE. 0. .AND. L .GE. 0.)
   20 FORMAT (3F10.4)
C     PRINT VALUES OF RESISTANCE, FREQUENCY AND INDUCTANCE
      WRITE (6,30) R,F,L
   30 FORMAT (3H1R=,F14.4,4H  F=,F14.4,4H  L=,F14.4)
C     INPUT STARTING AND TERMINATING VALUES OF CAPACITANCE, AND
C     INCREMENT
      READ (5,40) SC,TC,CI
      ASSERT (SC .GT. 0.)
      ASSERT (TC .GE. SC)
      ASSERT (CI .GT. 0.)
      ASSERT (CI .LE. TC - SC)
   40 FORMAT (3F10.6)
C     SET CAPACITANCE TO STARTING VALUE
      C = SC
      TEMPRES = R
      TMPFREQ = F
      TMPIND E= L
C     SET VOLTAGE TO STARTING VALUE
      V = 1.0
C     PRINT VALUE OF VOLTAGE
   50 WRITE (6,60) V
      ASSERT (V .GE. 1.0)
      ASSERT (C .GE. SC)
      ASSERT (V .LE. 3.0)
      TEMPI=0
   60 FORMAT (3H0V=,F5.0)
C     COMPUTE CURRENT AI
      ASSERT (F .NE. 0.)
      ASSERT (C .NE. 0.)
      ASSERT (R**2 + (6.2832*F*L - 1.0/(6.2832*F*C)) .NE. 0.)
   70 AI = E / SQRT(R**2 + (6.2832*F*L - 1.0/(6.2832*F*C))**2)
      ASSERT (AI .LE. 3./ABS(R))
      ASSERT (AI .LE. 3./ABS(6.2832*F*L - 1./(6.2832*F*C)))
      ASSERT (AI .GE. 1./(ABS(R) + ABS(6.2832*F*L - 1./(6.2832*F*C))))
      ASSERT (TEMPRES .EQ. R .AND.TMPFREQ .EQ. F .AND. TEMPIND .EQ. L)
      ASSERT (ABS(SC+TEMPI*CI-C) .LE. 1.E-4)
      TEMPI = TEMPI+1
      ASSERT (C .LE. TC)
C     PRINT VALUES OF CAPACITANCE AND CURRENT
      WRITE (6,80) C,AI
   80 FORMAT (3H0C=,F7.5,4H  I=,F7.5)
C     INCREASE VALUE OF CAPACITANCE
      C = C + CI
      IF (C .LE. TC) GOTO 70
C     INCREASE VALUE OF VOLTAGE
      V = V + 1.0
      ASSERT (C .EQ. SC)
C     STOP IF VOLTAGE IS GREATER THAN 3.0
      IF (V .LE. 3.0) GOTO 50
      STOP
      END
      PROGRAM FLOATPT
C.ASSERT=ON
C
C     TESTS FOR EXACT EQUALITY BETWEEN COMPUTED FLOATING POINT NUMBERS
C
C     SOURCE =   KERNIGHAN AND PLAUGER
C                THE ELEMENTS OF PROGRAMMING STYLE
C                PAGE 117.
C
C     RIGHT TRIANGLES
      LOGICAL RIGHT,DATA
      DO 1 K = 1,100
      ASSERT (K .LE. 100)
      READ (5,10) A,B,C
C     CHECK FOR NEGATIVE OR ZERO DATA
      DATA = A .GT. 0. .AND. B .GT. 0. .AND. C .GT. 0.
      IF (.NOT. DATA) GOTO 2
C     CHECK FOR RIGHT TRIANGLE CONDITION
       A = A**2
       B = B**2
       C = C**2
      ASSERT (A .GT. 0.)
      ASSERT (B .GT. 0.)
      ASSERT (C .GT. 0.)
      RIGHT = A .EQ. B + C .OR. B .EQ. A + C .OR. C .EQ. A + B
      ASSERT (RIGHT .OR. (ABS((B+C)-A) .GE. (A+B+C)*.001))
      ASSERT (RIGHT .OR. (ABS((A+C)-B) .GE. (A+B+C)*.001))
      ASSERT (RIGHT .OR. (ABS((A+B)-C) .GE. (A+B+C)*.001))
  1   WRITE (6,11) K,RIGHT
      CALL EXIT
C     ERROR MESSAGE
  2   WRITE (6,12)
      ASSERT (A .LE. 0. .OR. B .LE. 0. .OR. C .LE. 0.)
      STOP
 10   FORMAT (3F10.4)
 11   FORMAT (I6,L12)
 12   FORMAT (11H DATA ERROR)
      END
      PROGRAM INTEGR8
C.ASSERT=ON
C
C     INTEGRATES A POLYNOMIAL BY TRAPEZOIDAL APPROXIMATION
C
C     SOURCE =  KERNIGHAN AND PLAUGER
C               THE ELEMENTS OF PROGRAMMING STYLE
C               PAGE 116.
C
      AREA = 0.
      X = 1.
      DELTX = 0.1
      ASSERT (10.-X .GE. DELTX)
    9 Y = X**2 + 2. * X + 3.
      ASSERT (Y .GE. 6.)
      ASSERT (Y .LE. 123.)
      ASSERT (X .LT. 10)
      ASSERT (X .EQ. 1. .OR. Y .EQ. YPLUS)
      X = X + DELTX
      ASSERT (X .GE. 1.)
      ASSERT (X .LE. 10)
      YPLUS = X**2 + 2. * X + 3.
      ASSERT (YPLUS .GE. 6.)
      ASSERT (YPLUS .LE. 123.)
      TMPAREA = AREA
      AREA = AREA + (YPLUS + Y) / 2. * DELTX
      ASSERT (AREA .GT. TMPAREA)
      ASSERT (AREA .GT. 6. * (X-1.))
      ASSERT (AREA .LT. YPLUS * (X-1.))
      ASSERT (DELTX .EQ. 0.1)
      IF (X - 10.)9,15,15
   15 WRITE (6,7)AREA
      ASSERT (AREA .GT. 54.)
      ASSERT (AREA .LT. 9. * 123.)
      ASSERT (X .GE. 10.)
    7 FORMAT (E20.8)
      STOP
      END
      PROGRAM NUMALPH
C.ASSERT=ON
C
C      A PROGRAM WITH A SUBTLE INITIALIZATION ERROR
C
C      SOURCE = KERNIGHAN AND PLAUGER
C               THE ELEMENTS OF PROGRAMMING STYLE
C               PAGE 80 OF FIRST EDITION
C      AUGMENTED TO PRODUCE SOME OUTPUT   7/11/78   REG MEESON
C
       DIMENSION NUM(80),NALPHA(80)
       DATA NBLANK /1H /
     1 READ (5,101,END=99) NALPHA,NUM
       ASSERT ((.ALL. I .IN. (1,80)
     *   (NALPHA(I) .GE. 1H0 .OR. NALPHA(I) .EQ. 1H )))
       ASSERT ((.ALL. I .IN. (1,80)
     *   (NALPHA(I) .LE. 1H9 .OR. NALPHA(I) .EQ. 1H )))
   101 FORMAT (80A1,T1,80I1)
       WRITE (6,102) NALPHA,NUM
   102 FORMAT (11H INPUT DATA / 1H0,80A1 / 1H  ,80I1)
       N = 0
       DO 30 I = 1,80
       IF (NALPHA(I) .EQ. NBLANK) GOTO 30
       N = N + 1
       TEMPSUM = NSUM
       ASSERT (N .LE. I)
       NSUM = NSUM + NUM(I)
       ASSERT (NSUM .GE. TEMPSUM)
       ASSERT (NSUM .LE. N * 9)
       ASSERT (NSUM .GE. 0)
    30 CONTINUE
       WRITE (6,103) N,NSUM
       ASSERT (NSUM .LE. N * 9)
       ASSERT (NSUM .GE. 0)
       ASSERT (N .LE. 80)
   103 FORMAT (30H0THE NUMBER OF DIGITS FOUND IS, I3 /
     $         29H AND THE SUM OF THE DIGITS IS,  I4 )
       GOTO 1
    99 STOP
       END
      PROGRAM SINEFCN
C.ASSERT=ON
C
C     DRIVER PROGRAM TO TEST THE DOUBLE PRECISION SINE FUNCTION
C     REG MEESON   7/11/78
C
      DOUBLE PRECISION  SINE,DSIN,DBLE,REF,VAL,E
      REAL  X
C
      WRITE (6,100)
   10 READ (5,110) X,E
      WRITE (6,120) X,E
      IF (E .EQ. 0) STOP
      REF = DSIN(DBLE(X))
      VAL = SINE(X,E)
      WRITE (6,130) REF,VAL
      GOTO 10
C
  100 FORMAT (26H SINE FUNCTION TEST DRIVER //)
  110 FORMAT (F10.4,D10.2)
  120 FORMAT (3H X=,F10.4,7H     E=,D20.12)
  130 FORMAT (1H ,45X,4HREF=,D20.12,9H     VAL=,D20.12)
C
      END
DOUBLE PRECISION FUNCTION SINE(X,E)
C
C     SOURCE = KERNIGHAN AND PLAUGER
C              THE ELEMENTS OF PROGRAMMING STYLE
C              PAGE 101.
C
C     THIS DECLARATION COMPUTES SINE(X) TO ACCURACY E
      DOUBLE PRECISION E,TERM,SUM
      REAL  X
      ASSERT (ABS(X) .LE. 3.14159)
      ASSERT (E .GT. 0)
      TERM = X
      DELT = SUM
      DO 20 I = 3,100,2
      TERM = TERM*X**2/(I*(I-1))
      ASSERT (I .EQ. 3 .OR. ABS(TERM) .LT. TMPTERM)
      TMPTERM = ABS(TERM)
      IF (TERM .LT. E) GOTO 30
      TEMPSUM = SUM
      SUM = SUM + (-1**(I/2)) * TERM
      TEMPDEL = DELT
      DELT = SUM - TEMPSUM
      ASSERT (TEMPDEL * DELT .LT. 0.)
   20 CONTINUE
   30 SINE = SUM
      ASSERT (TERM .LT. E)
ASSERT (SINE .GE. -1. .AND. SINE .LE. 1.)
ASSERT (ABS(SINE) .LT. ABS(X) .OR. X .EQ. 0.)
RETURN
END
```

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.