U.S. DEPARTMENT OF COMMERCE
National Technical Information Service

AD-A016 668

STRUCTURED PROGRAMMING SERIES.

VOLUME XV. VALIDATION AND VERIFICATION STUDY

IBM FEDERAL SYSTEMS CENTER

PREPARED FOR

ROME AIR DEVELOPMENT CENTER

ARMY COMPUTER SYSTEMS COMMAND

22 MAY 1975

RADC-TR-74-300, VOLUME XV 314164 MAY 75

STRUCTURED PROGRAMMING SERIES

ADA016668

VALIDATION & VERIFICATION STUDY

PRICES SUBJECT TO CHANGE

APPROVED FOR PUBLIC RELEASE.
DISTRIBUTION UNLIMITED

CO-SPONSORED BY

U.S. ARMY
COMPUTER SYSTEMS COMMAND
FORT BELVOIR, VIRGINIA

U.S. AIR FORCE
AIR FORCE SYSTEMS COMMAND
ROME AIR DEVELOPMENT CENTER
GRIFFISS AIR FORCE BASE, N.Y.

PRICES SUBJECT TO CHANGE

Reproduced by
NATIONAL TECHNICAL INFORMATION SERVICE
US Department of Commerce
Springfield, VA. 22151

This report has been reviewed by the Office of Information, RADC, and approved for release to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

NOTE: Volumes I, II, III, IV, V, VI, VII, Addendum to Volume VII, VIII, IX, X, XI, and XIV of the Structured Programming Series are the only volumes which have been published up to this time.

Copies of these and subsequent volumes of the Structured Programming Series may be obtained from the

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161

by referencing RADC-TR-74-300.

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

Do not return this copy.
Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
RADC-TR-74-300, Volume XV		

4. TITLE (and Subtitle)
STRUCTURED PROGRAMMING SERIES (Volume XV)
Validation and Verification Study

5. TYPE OF REPORT & PERIOD COVERED
Final Technical Report

6. PERFORMING ORG. REPORT NUMBER
N/A

7. AUTHOR(s)
Ronald L. Smith

8. CONTRACT OR GRANT NUMBER(s)
F30602-74-C-0186

9. PERFORMING ORGANIZATION NAME AND ADDRESS
IBM Corporation, Federal Systems Center
18100 Frederick Pike
Gaithersburg, Maryland 20760

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
A00H
55500803

11. CONTROLLING OFFICE NAME AND ADDRESS
Rome Air Development Center (ISIS)
Griffiss Air Force Base, New York 13441

12. REPORT DATE
22 May 1975

13. NUMBER OF PAGES
89

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:
Donald L. Mark (ISIS)
AC 315 330-4875

USACSC Project Officer:
Captain John C. Carrow
AC 703 664-4235

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

Verification
Validation
Inspection
Testing
Certify
Prove

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

This volume reports on techniques currently used for verifying and validating computer programs and software systems. It also contains an analysis of the effect that structured programming technology will have on these techniques. This analysis addressed all phases of software development - definition, design, implementation, and evaluation.

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

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

STRUCTURED PROGRAMMING SERIES

VOLUME XV

Validation and Verification Study

Final Report

Ronald L. Smith

International Business Machines Corporation
Federal Systems Center
18100 Frederick Pike
Gaithersburg, Maryland 20760

May 22, 1975

Approved for public release;
distribution unlimited.

Produced Under
U.S. Air Force (RADC)
Contract F30602-74-C-0186
Co-sponsored by USACSC

ia

FOREWORD

This report was produced in response to Task 4.1.15 in the Statement of Work for the Structured Programming System under contract number F30602-74-C-0186. The report is delivered to RADC in accordance with item A00H of the Contract Data Requirements List.

The report was prepared by Ronald L. Smith with significant contributions by Ms. M. C. Blakebrough and Mr. J. J. Naughton, and constructive criticism by Mr. D. W. Daetwyler, Mr. T. M. Kraly, Mr. J. W. Patterson, Mr. N. Tinanoff and Mr. J. T. Trimble.

Contributions were also made by the following RADC personnel: Mr. F. J. Tomaini, Mr. R. Nelson, and Mr. D. L. Mark, Project Engineer; and the following USACSC personnel: Mr. G. Pelled, Mr. H. E. Kody, and Captain J. C. Carrow, Project Officer.

This report is one of a set called the Structured Programming Series. The objective of this set is to provide information, guidelines and standards as appropriate to facilitate the adoption and use of structured programming technology in the acquisition and development of software. The Structured Programming Series consists of the following volumes:

Volume I	-	Programming Language Standards
Volume II	-	Precompiler Specifications
Volume III	-	Precompiler Program Documentation
Volume IV	-	Data Structuring Study
Volume V	-	Programming Support Library Functional Requirements
Volume VI	-	Programming Support Library Program Specifications
Volume VII	-	Documentation Standards
Volume VIII	-	Program Design Study
Volume IX	-	Management Data Collection and Reporting
Volume X	-	Chief Programmer Team Operations Description
Volume XI	-	Estimating Software Project Resource Requirements
Volume XII	-	Training Materials
Volume XIII	-	Final Report
Volume XIV	-	Software Tool Impact
Volume XV	-	Validation and Verification Study

APPROVED:

GEORGE M. SOKOL
Deputy for Engineering
US Army Computer Systems Command

CARLO P. CROCETTI
Chief, Plans Office
Rome Air Development Center

ABSTRACT

This volume reports on techniques currently used for verifying and validating computer programs and software systems. It also contains an analysis of the effect that structured programming technology will have on these techniques. This analysis addresses all phases of software development - definition, design, implementation, and evaluation. There are two major conclusions of the study. First, a majority of software projects rely almost entirely on computer based testing as the method of verifying and validating software. Second, structured programming technology has facilitated manual based verification techniques such as design verification and code verification, and thus, increased usage of these techniques should be encouraged.

EVALUATION

The objective of this effort is to design and develop a detailed set of guidelines for a Structured Programming System (SPS) that will be used to create a complete environment for the acquisition and development of software. Contract F30602-74-C-0186 will serve to transfer IBM's present technology in Structured Programming (SP), Top Down Development, Chief Programmer Teams (CPT), and Structured Programming Libraries (SPL) to the Air Force for further development. The technology transfer will be realized by studying and refining specific software production task areas.

This report, which is one of a set called the Structured Programming Series, describes the impact of structured programming technology on validation and verification of software.

Structured programming (SP) technology is providing the opportunity for improved techniques for determining correctness at both the program (verification) and system (validation) level. Some classical techniques, such as the use of driver to exercise code, are becoming obsolete while others, such as code reading and desk checking, are becoming more important. Initially, all elements of the SP technology were considered on all types of software development. The primary emphasis, however, was expected to be on the impact of top down structured programming on large scale development projects.

A survey of the technical literature in the area of software verification and validation was performed. IBM Federal System Division experience in program verification and validation on selected structured programming projects was analyzed. Finally, technical judgments regarding the impact of structured programming technology on the verification and validation of computer programs were made.

DONALD L. MARK
Project Engineer
Software Sciences Section

TABLE OF CONTENTS

Section		Page
1	INTRODUCTION	1-1
1.1	Background	1-1
1.2	Report Organization	1-1
1.3	Conclusions	1-2
1.4	Recommendations	1-3
2	DEFINITION FRAMEWORK FOR VERIFICATION AND VALIDATION	2-1
2.1	Terminology History	2-1
2.2	Definitional Framework	2-1
2.3	Related Testing Activities	2-3
3	VERIFICATION AND VALIDATION TECHNIQUES SURVEY	3-1
3.1	Introduction	3-1
3.2	Verification Techniques	3-4
3.3	Validation Techniques	3-17
4	STRUCTURED PROGRAMMING PROJECT SURVEY	4-1
4.1	Introduction	4-1
4.2	Mitsubishi Bank New System - Japan	4-1
4.3	Ground Support Simulation Computer for ASTP - Houston, Texas	4-3
4.4	Advanced Control System (ACS) - Houston, Texas	4-5
4.5	Earth Resources Interactive Processing System (ERIPS) - Houston, Texas	4-6
4.6	FAA National Airspace System Support Software (NASCOR, ACEUTE) - Atlantic City, New Jersey	4-8
4.7	Resource Monitoring System Force Status - Rosslyn, Virginia	4-9
4.8	Multi-Optical Sensor Simulation (MOSS) - Westlake, California	4-11
4.9	Test Control Computer Support Software (TCCSS) - Gaithersburg, Maryland	4-13
4.10	TRIDENT AN/BQQ-6 Sonar Software - Manassas, Virginia	4-14
4.11	Energy Management System (EMS) - Houston, Texas	4-16
5	STRUCTURED PROGRAMMING TECHNOLOGY (SPT) IMPACT ON VERIFICATION AND VALIDATION	5-1
5.1	Introduction	5-1
5.2	Techniques	5-1
5.3	SPT Impact on Techniques Described in Section 3	5-9
Appendix A	SOFTWARE DEVELOPMENT CYCLE	A-1
Appendix B	STRUCTURED PROGRAMMING TECHNOLOGY FUNDAMENTALS	B-1
Appendix C	REFERENCES	C-1
Appendix D	BIBLIOGRAPHY	D-1

ILLUSTRATIONS

Figure		Page
2-1	Framework for Verification and Validation	2-2
3-1	Use or Potential Use of Verification and Validation Techniques	3-2
3-2	Advanced Execution Analysis Approach	3-11
3-3	Example of Network as Path Analysis	3-13
3-4	Software Confidence Calculations	3-16

Section 1

INTRODUCTION

1.1 BACKGROUND

Verification and validation of digital computer programs has been a difficult and costly activity of the software development process. Structured programming has been ascribed an advantage in verification and validation by its early proponents.

In recognition of these factors, Task 4.1.15 was included in the Statement of Work for contract number F30602-74-C-0186. The objective of this task is to determine what verification and validation techniques are currently being used for determining program and system correctness and how these techniques are impacted by structured programming technology.

In pursuit of this objective, the following subtasks were performed:

- o Review of available literature related to verification and validation techniques of software.
- o Review of selected IBM Federal Systems Division (FSD) projects which use structured programming technology to determine experience with software verification and validation.
- o Identification and examination of existing verification and validation techniques.
- o Determination of the potential impact of structured programming technology on the traditional techniques used to verify and validate software.

This report presents the results of these subtasks.

1.2 REPORT ORGANIZATION

Section 1 contains background information, conclusions and recommendations of the study.

Section 2 contains the definitional framework for verification and validation and brief discussions on related testing activities.

Section 3 describes some of the verification and validation techniques currently used on software development projects and those with future potential. The utility, advantages, and disadvantages of each technique are also described.

Section 4 contains the results of the survey of software development projects (current or recently completed) and discussions on the verification and validation techniques used on each project.

Section 5 describes two verification techniques that are usually associated with structured programming technology. This section also discusses how structured programming technology affects the verification and validation techniques described in Section 3.

Appendix A describes the software development cycle.

Appendix B contains a discussion and definition of structured programming technology.

Appendix C contains the references used in this report.

Appendix D contains the bibliography.

1.3 CONCLUSIONS

Significant findings of the study are:

a. A majority of software development projects rely almost entirely on computer based testing as the method of verifying and validating software. However, there is no widely accepted technique for verifying and validating software.

b. Structured Programming Technology (SPT) facilitates verification and validation of software, but at this time insufficient data is available to support specific approaches using the techniques described in this report.

c. SPT provides the top down testing technique resulting from using top down programming (TDP). This verification technique assists in producing more reliable software at a lower cost.

d. SPT has a direct positive affect on two manual based techniques, i.e., design verification and code verification; and one computer based technique, drivers. The SPT impact on the manual based techniques facilitates the early detection and removal of errors and thus, reduces the cost of developing and maintaining software. SPT also reduces the number of drivers required which results in a cost savings of software development.

e. SPT has an indirect positive affect on four other verification and validation techniques. They are execution analysis, automated network and path analysis, functional testing, and design simulation.

1.4 RECOMMENDATIONS

The following recommendations will achieve a significant improvement in the production and maintenance of software even though the SPT affect on verification and validation cannot be quantified:

a. Introduce the concepts of top down programming on all future new software procurements employing the associated top down testing technique to reduce the cost and increase the reliability of software. Top down testing should be used as the major verification technique during the computer based testing of a software development project.

b. Encourage the use of manual based verification techniques which are facilitated by using SPT. Techniques such as design verification and code verification should be used to find errors early in the software development process.

Section 2

DEFINITION FRAMEWORK FOR VERIFICATION AND VALIDATION

2.1 TERMINOLOGY HISTORY

The terms verification and validation along with several related terms are used to refer to the error finding and evaluation activities of software development. Terms used include verify, prove, validate, certify, debug, test, and inspect. A review of the literature demonstrated the semantic difficulty in this area. During the early period of software development, the two terms debugging and testing were most frequently used. Fred Gruenberger [1] states that these two terms were used synonymously until 1957 when program testing was distinguished from debugging by Charles Baker of the RAND Corporation. The distinction made was that debugging starts with known errors and attempts corrections, testing measures how well the specifications are met. During the period from 1957 to present, many writers continued to confuse the terms testing and debugging. In the early sixties, the terms validate and verify started appearing in the literature, and by the seventies, other terms such as certify, prove, and inspect began to appear in the literature.

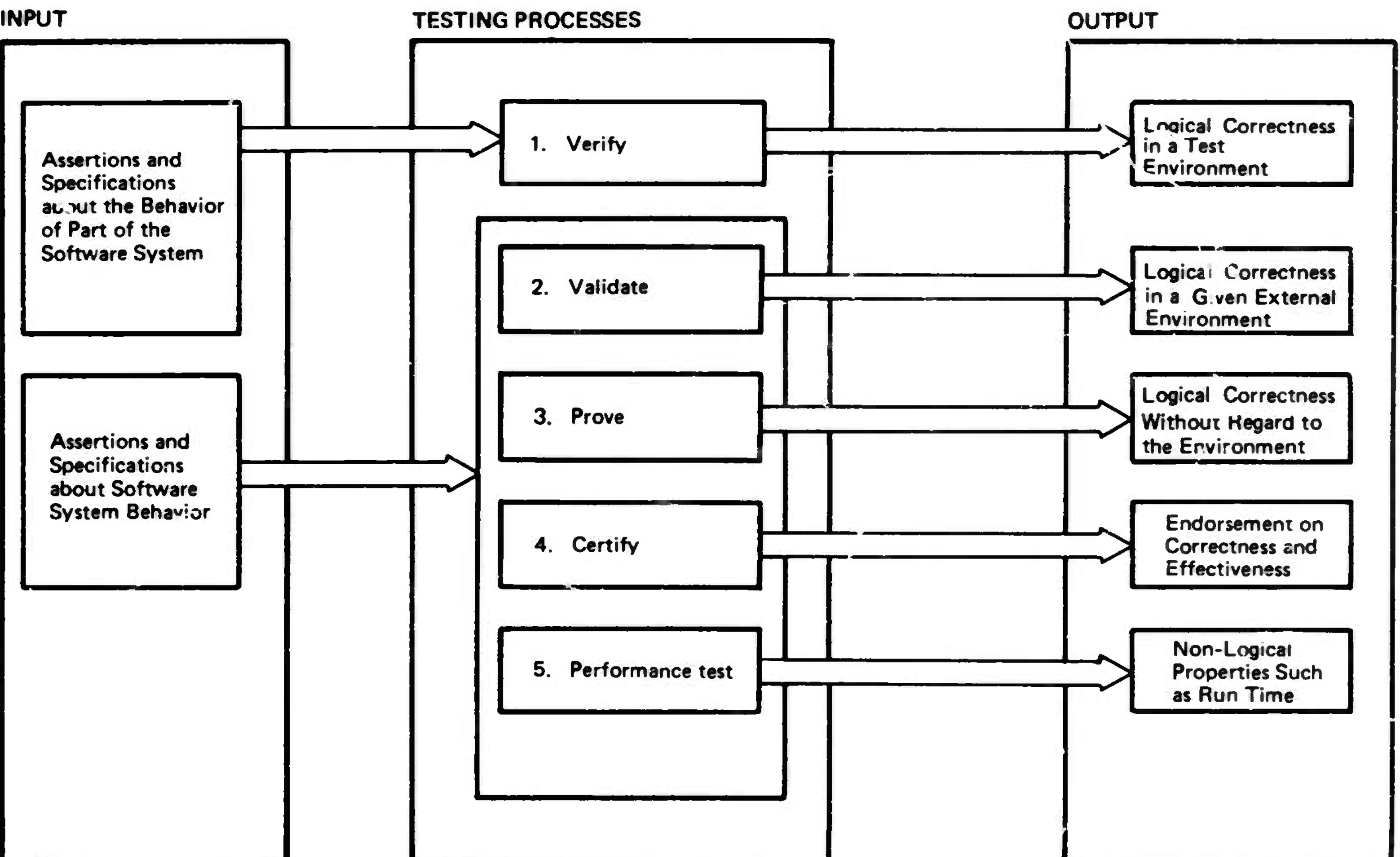
This section attempts to eliminate some of the semantic ambiguity by defining a testing framework that will be used throughout this report. Using this framework, several testing activities (i.e., correctness proof, certification, and performance test) that are outside the scope of this study will also be briefly addressed.

2.2 DEFINITIONAL FRAMEWORK

The framework defined here and shown in Figure 2-1 is similar to the one defined by William Hetzel [2]. This figure shows the testing activity. The specification set or assertions which describe the software behavior are shown as an input to all of the testing activities. Five major test activities are defined in the process boxes. Inputs to a process box are suggestive of the information used by the process in the box while outputs on the right suggest the information obtained. Debugging is not included in the definition of testing as defined earlier. This study concentrated on processes 1 and 2 which are discussed in more detail. The definitions for these two processes are presented below. Manual and computer based techniques described in these two processes are described in Section 3.

2.2.1 Verification Definition

Verification is the process of determining whether the results of executing the software product in a test environment agree with the specifications.



2-1. **Framework for Verification and Validation**

Verification is usually only concerned with the software's logical correctness (i.e., satisfying the functional requirements) and may be a manual or a computer based process (i.e., testing software by executing it on a computer).

2.2.2 Validation Definition

Validation is the process of determining whether executing the system (i.e., software, hardware, user procedures, personnel) in a user environment causes any operational difficulties. Validation is more difficult than the verification process since it involves questions of the completeness of the specification and environment information. There are both manual and computer based validation techniques.

2.3 RELATED TESTING ACTIVITIES

2.3.1 Correctness Proof

Correctness proof of a computer program is most often defined as the technique of proving mathematically that a given program is consistent with a given set of specifications. This process can be accomplished by manual methods or by program verifiers requiring manual intervention. The latter concept evolved from a dissertation by Jim C. King at Carnegie-Mellon University in September 1969 that was later presented in a book [3].

Program verifiers make use of formal specifications of the program's intent that are written in a formal assertion language. The correctness proof process then consists in analyzing the actions carried out by the program and checking, usually by proving mathematical theorems, that the output assertions will be satisfied whenever the input data meets the conditions specified by the input assertions. Several research papers have been written on program verifiers and a few of these are listed under References. The most prolific writers on this subject are: Donald I. Good [4, 5], Ralph London [5], Bernard Elspas [6], and Larry Robinson [7].

Significant progress has been made in this area since 1970, but most authors agree that program verifiers are a long way from being able to handle practical programs of substantial size written in conventional programming languages. Boyer, Elspas, and Levitt [6] list five main obstacles to formally verifying realistic programs. They are:

a. The necessity (in most systems) for inventing intermediate assertions (e.g., inductive assertions, Floyd predicates), and the difficulty experienced by the programmer-verifier in coming up with these assertions.

b. The difficulty in framing input and output assertions that adequately express the program's intent.

c. The lack of sufficiently rich assertion languages in which to express these input/output assertions and intermediate assertions for programs covering a wide variety of data types and functional primitives.

d. Technical limitations in present-day theorem-proving techniques such as inadequate speed and large storage requirements.

e. An apparent need for considerable programmer intervention in the theorem-proving process.

2.3.2 Certification

Certification as discussed by Hetzel [2] carries the connotation of an authoritative endorsement and seems to imply testifying in writing that the program is of a certain standard or quality. Certification usually implies the existence of an independent quality control group. Reifer [8] states that certification extends the process of verification and validation to an operational environment and involves acceptance testing of the overall system. Keirstead and Parker [9] define basic requirements of a software system certification. They are:

a. Develop means for determining the adequacy of software specifications.

b. Develop methods for calculating a reliability measure for a component software module.

c. Develop procedures for combining component measures to derive a measure of software system reliability.

Certification and several other areas related to certification were not the primary emphasis of the study and are not further discussed in this report. The related areas and references contained in the bibliography can be summarized as Software Reliability [10, 11, 12, 13, 14], Software Complexity [11, 15], and Quality Assurance [16, 17]. For a comprehensive bibliography on certification and the other related activities, reference the bibliography in the book, Program Test Methods, edited by William C. Hetzel [18].

2.3.3 Performance Testing

Performance testing of software is defined as the evaluation of nonlogical properties (i.e., computer run time, resource utilization) of a software system. Performance is measured in terms of the amount of resources required by a software system to produce a result.

The literature for the most part does not address performance testing independently from functional testing. Hetzel [2] and Merten and Teichroew [19] do address performance testing as an independent testing activity. The latter paper describes how problem statement languages can assist in the performance testing of software.

7<

The verification and validation techniques (discussed in Sections 3, 4, and 5) are primarily oriented toward functional testing; however, a few of these techniques are indicated as also being suitable for performance testing.

Section 3

VERIFICATION AND VALIDATION TECHNIQUES SURVEY

3.1 INTRODUCTION

One of the most difficult, time consuming aspects of software development is comprehensive verification and validation of the system's capability to perform intended functions. The basic purpose of verifying and validating is to ensure that a software product will perform its intended function at the time those functions are needed by the user.

Most large computer software systems are never completely verified and validated. For such a claim to be valid would require the successful execution of an astronomical number of tests designed to test every logical and illogical combination of data, or timing environment, through every logical data path in the system. Such a degree of testing is usually neither feasible nor practical. Therefore, the practical approach usually taken is to ensure that every logical path is tested with combinations of data which include the nominal and any reasonably expected deviations from the nominal.

Most of the verification and validation techniques, described in the literature and used by the software projects surveyed during this study, address this practical approach with computer based testing and not manual based testing. Computer based testing and manual based testing are discussed in subsections 3.1.1 and 3.1.2.

The classification of the techniques as either a verification technique or a validation technique was a technical judgement and was not necessarily indicated in the literature.

Section 3 does not describe all the techniques but does describe (in order of decreasing usage) the traditional techniques which have been used most often followed by those judged to have near term practical application value. An example of a validation technique not included in this report is the DoD COBOL compiler validation system described by Baird [20]. This technique, while extremely useful in validating COBOL compilers, is limited in the validation of other software systems. Two verification techniques not described in Section 3 usually associated with structured programming technology are described in Section 5. An overview of the verification and validation techniques and their use or potential use during the various phases of the software development cycle* is depicted in Figure 3-1.

---

* The software development cycle is discussed in Appendix A.

Verification and Validation Techniques	Software Development Phases			
	Definition	Design	Implementation	Evaluation
Drivers			X	
Test Data Bases			X	X
Design Verification		X	X	
Execution Analysis			X	X
Automated Network or Path Analysis			X	
Statistical Prediction		X	X	X
Functional Testing			X	X
Design Simulation	X	X	X	
Design Validation	X	X		
Matrix Analysis and Problem Statement Languages	X	X		
Top Down Programming			X	
Code Verification			X	

Figure 3-1. Use or Potential Use of Verification and Validation Techniques

3.1.1 Computer Based Testing

Computer based testing can be broadly divided into two types: informal testing and formal testing. The basic difference between these types originates from the documentation requirements. Informal testing utilizes internal test documentation control and procedures; formal testing is conducted in accordance with customer approved test plans.

Informal testing usually is designed to be development group testing and requires no formal customer approval. Informal testing usually begins when the first program unit is coded and continues throughout the system implementation phase of software development (refer to Appendix A for the description of the software development cycle). Terms used in the literature to describe this testing include:

- o Unit Testing
- o Subsystem Testing
- o Integration Testing
- o Component Testing
- o Development Testing.

Formal testing is the testing performed in accordance with customer-approved test plans. This type of testing verifies that the software system is operating according to the requirements of the development specifications. Formal testing is usually performed during the system evaluation phase of software development. Terms used in the literature to describe this testing include:

- o System Integration Testing
- o Prototype Testing
- o System Testing
- o Acceptance Testing.

3.1.2 Manual Based Testing

Manual based testing is usually directed at evaluating both the design and the product (i.e., programs and documentation). The design is usually evaluated from documents containing information such as functional requirements, system specifications, and program specifications. The product evaluation usually involves review of the computer programs and the documentation describing the programs or system.

3.2 VERIFICATION TECHNIQUES

3.2.1 Drivers

3.2.1.1 Description

A driver program is superfluous (throw-away) code needed to perform the unit testing and lower levels of integration testing in a bottom up software development effort. Frequently, test drivers were used to test the entire software system. The literature indicates that most traditional software development projects used this technique in one form or another. Several authors [21, 22, 23, 24, 25] describe driver programs that were developed for the automatic generation of test cases and for driving other programs. These general purpose driven programs were developed in an attempt to increase programmer productivity and quality of testing.

3.2.1.2 Use

Drivers are a computer based informal testing technique and are used in conjunction with other techniques almost exclusively during the system implementation phase of a software development project. This appears to be the most frequently used verification technique. It was used on seven of the ten programming projects described in Section 4.

3.2.1.3 Advantages

The major advantages of this technique are:

a. The testing of critical components can be emphasized.

b. The I/O functions are most often required early in the software development and these can easily be handled with driver programs.

3.2.1.4 Disadvantages

The major disadvantages of this technique are:

a. Cost of developing the drivers which in most cases are discarded and not delivered to the customer.

b. Errors frequently exist in drivers which further impede the testing process.

c. Drivers are usually written by the same individuals who write the component being tested. Thus, they are likely to contain the same invalid assumptions about the component interface.

3.2.2 Test Data Bases

3.2.2.1 Description

A test data base is a collection of data stored on a computer peripheral device (e.g., tape, disk) that closely matches the "real" data base. Ideally, a test data base should be identical to a real data base but usually it only provides representative data. There are many test data generator programs available, and some of these are tabulated in [25]. These programs generate tape or disk files with random or sequential or user specified data values.

3.2.2.2 Use

The literature indicates the frequent use of this technique for the development of large software systems. Test data bases were used on the New York Times Project as described in [26], on the Federal Aviation Administration National Air Space System as described in [27], and on nine of the ten programming projects described in Section 4. This technique is usually used in conjunction with one or more of the other verification techniques. This testing technique is computer based and informal, used almost exclusively during the system implementation phase of a software development project.

3.2.2.3 Advantages

The major advantages of this technique are:

a. The real data base is protected.

b. The testing can proceed faster with less confusion.

3.2.2.4 Disadvantages

The major disadvantages of this technique are:

a. The high cost of creating a test data base.

b. Test data bases do not always contain adequate data types or a sufficient volume of data.

3.2.3 Design Verification

3.2.3.1 Description

This technique is defined as the examination or inspection of a software specification for the purpose of finding design errors. Other terms used in the literature to describe this technique or variations of this technique

include design review, design inspection, specification testing, paper testing, walk-through, structured walk-through, and preliminary design review. Two approaches to the design verification technique are presented below.

Buckley [28] describes a Preliminary Design Review (PDR). He states that the purpose of each PDR was to examine the preliminary design of a particular module of software. Each PDR was to be preceded with extensive technical notes including flowcharts provided by the contractor. After an in-house review, the customer and the contractor were to meet, discuss the approach on the specific software module in detail, ensure this approach was consistent with the contract, and jointly resolve any discrepancies.

Fagan [29] describes design inspections. His I_1 design inspection is summarized below under the three headings: "Inspection Team," "Outline of the Inspection Procedure," and "Examples of What to Examine When Looking for Errors."

a. Inspection Team

1. Moderator-The key person in a successful inspection is the moderator. He need not be a technical expert, but he must manage the inspection team and provide leadership. He must use personal sensitivity, tact, and drive in balanced measures. His use of the strengths of team members should produce a synergistic effect larger than their number. He is The Coach.

2. Designer-The programmer responsible for producing the program design.

3. Coder/Implementor-The programmer responsible for coding the design.

4. Tester-The programmer responsible for writing and/or executing test cases or otherwise testing the product of the designer and coder.

In the event that the coder is also the designer, he will function in the designer role, and another programmer from some related or similar program will perform the role of coder.

In the event that the same person will design, code, and test the product code, the coder role should be filled as described above. Another programmer, preferably with testing experience, should fill the role of tester.

b. Outline of the Inspection Procedure

1. Overview-Documentation of the design is distributed to all inspection participants. The designer describes the overall area being addressed, and then the specific area he has designed in detail such as logic, paths, and dependencies. The entire inspection team participates in the overview.

2. Preparation-Participants, using the design documentation, try to understand the design, its intent, and logic.

3. Inspection-The coder describes how he will implement the design. Every piece of logic is covered at least once and every branch is taken at least once. The entire team participates in this phase.

4. Followup-It is imperative that every issue, concern, and error be entirely resolved at this level. Errors can be 10 to 100 times more expensive to fix if found later in the process (programmer time only, machine time not included).

c. Examples of What to Examine When Locking for Errors

1. Missing

(a) Are all constants defined?
(b) Are all unique values explicitly tested on input parameters?
(c) Are values stored after they are calculated?
(d) Are all defaults checked explicitly tested on input parameters?
(e) If character strings are created, are they complete?
(f) Are all delimiters shown?
(g) If a keyword has many unique values, are they all checked?
(h) If a queue is being manipulated, can the execution be interrupted; if so, is queue protected by a locking structure?
(i) Can queue be destroyed over an interrupt?
(j) Are registers being restored on exits?
(k) Are all operands tested in macro?
(l) Are all keyword related parameters tested in service routines?
(m) Are queues being held in isolation so that subsequent interrupting requestors are receiving spurious returns regarding the held queue?
(n) Should any registers be saved on entry?
(o) Are all increment counts properly initialized (0 to 1)?

2. Wrong

(a) Are absolutes shown where there should be symbolics?
(b) On comparison of two bytes, should all bits be compared?
(c) On built data strings, should they be character or hex?
(d) Are internal variables unique or confusing if concatenated?

3. Extra

(a) Are all blocks shown in design necessary or are they extraneous?

3.2.3.2 Use

This technique or variations of this technique is frequently used during the system design and system implementation phases of a software development project. It is a manual technique used on projects described by Corrigan [30], Buckley [28], Fagan [29], and Scherr [31]. Fagan presents data on the net savings of using inspections. The design verification technique was also used on six of the ten programming projects described in Section 4.

3.2.3.3 Advantages

The advantages of this technique are:

a. The error rework cost (which is a significant variable in product cost) decreases due to the early detection of errors.

b. The quality of the product is improved.

c. An increase in programmer productivity is achieved due to a positive psychological effect that design verification has on programmers.

3.2.3.4 Disadvantages

The disadvantages of this technique are:

a. The lack of enthusiasm by the people performing a design verification. Design verifications are considered by some people to be tedious and boring.

b. Some errors found during a design verification would be more easily found by the compiler after the program is coded. This is usually the result of reviewing an overly detailed design.

3.2.4 Execution Analysis

3.2.4.1 Description

This technique is defined as the automated monitoring of the computer based software testing activities, collecting data from these testing activities, and subsequently predicting, by manually analyzing the data, the duration and cost of testing, and the quality of the software product. Other terms used in the literature referring to this technique or variations of this technique include code analyzer, code auditor, program evaluator, and product assurance evaluator.

There are two areas of data collections on testing. First, error data from past projects is collected and analyzed. This data can be used to evaluate costs and trends in testing. Second, the data is collected while a project

is being developed and tested. This technique, as described in the literature, is usually used in the second area.

There are two approaches for implementing this technique. One approach is to design data collecting tools into the system during the system design phase and develop these tools as an integral part of the end product. The other approach is to develop stand-alone tools or system to collect and report data. Several tools using the latter approach to the execution analysis technique are described below.

3.2.4.1.1 Program Evaluator and Tester (PET)

Stucki [32, 33, 34] describes a tool - Program Evaluator and Tester (PET) - that uses the execution analysis technique. This tool "instruments" an application software package by inserting the software equivalent of sensors into the package. Each time the software package under test performs a significant event, the occurrence of this event is recorded. He describes the recorded data as follows:

a. The number and percentage of all potential executable source statements which were executed one or more times.

b. The number and percentage of those program branches taken.

c. The number and percentage of those subroutine calls which were executed.

d. The number of times each subroutine was called, together with a list of those subroutines that were never entered.

e. Relative timing on the subroutine level.

f. Specific data associated with each executable source statement.

    1. Detailed execution counts.

    2. Detailed branch counts on all IF and explicit branches or GOTO statements.

    3. Optional data range values (min/max/first/last) on assignment statements.

    4. Optional min/max ranges on DO loop control variables.

A more advanced variation of this instrumentation tool is described by Stucki and Foshee [35]. They describe a new series of automated functions that are being designed and implemented. They further describe the incorporation of design verification criteria directly into evolving systems through a powerful assertion capability. The scope of the assertions presented encompasses the entire life cycle of a programming system from initial program design through operation and maintenance. A preprocessor examines a source program

and inserts additional source statements to gather pertinent statistics during program execution. A postprocessor matches statistics generated during program execution with individual source program statements to produce an annotated program listing and summary report. This advanced approach is shown in Figure 3-2. They have identified three types of automated instrumentation of programs. They are:

a. Monitoring source statements execution and branch conditions.

b. Verification of assertions on data characteristics and program behavior.

c. Monitoring range of values assumed by scalar variables, arrays, and subscripts.

3.2.4.1.2 Product Assurance Confidence Evaluator (PACE)

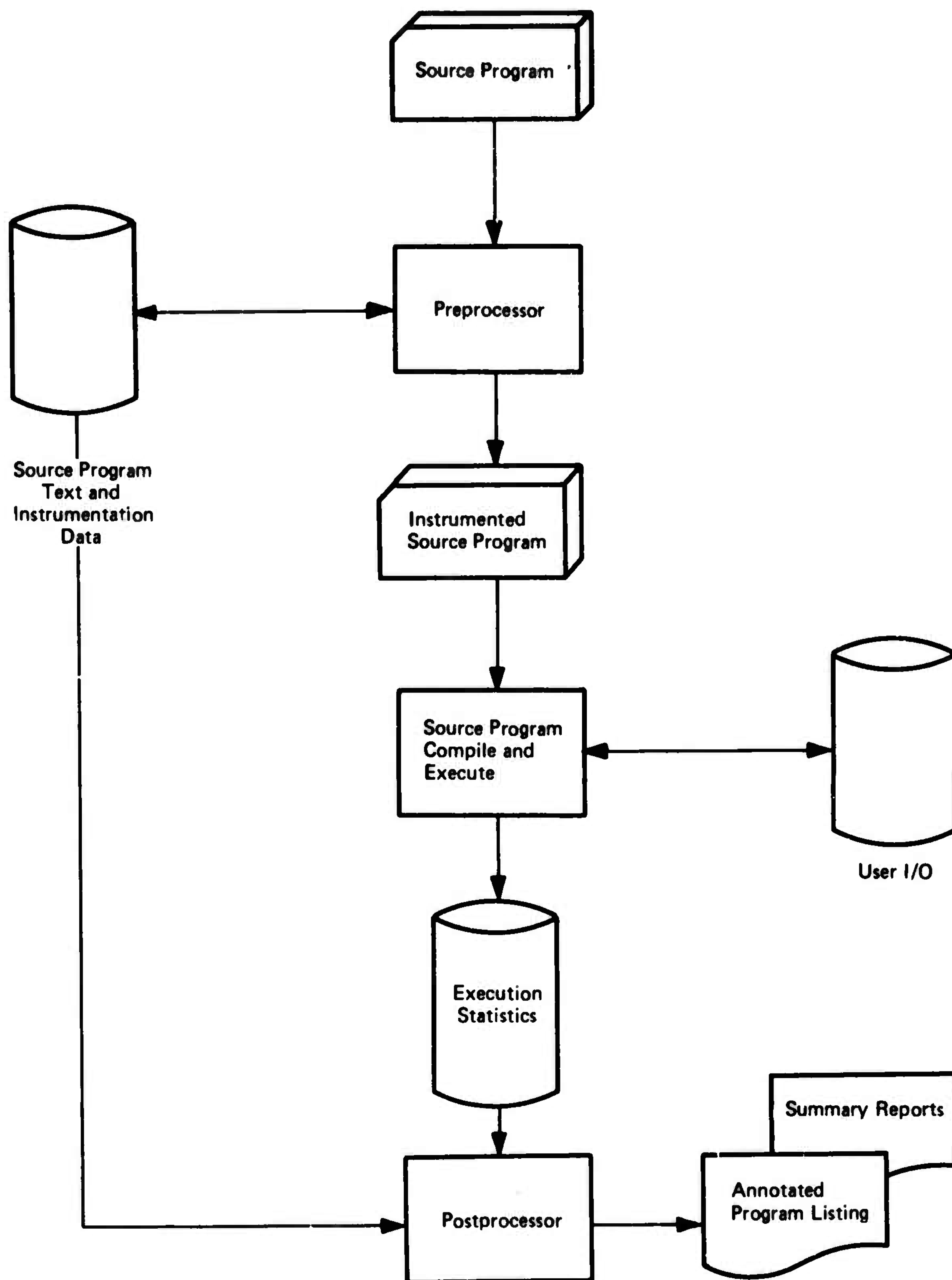
This system, comprised of automated tools, is described by Brown and Hoffman [21] and Nelson [36]. This system is designed to assist in the planning, production, execution, and evaluation of computer program testing. Two components of the PACE system, AUDIT and PATH, define all logic paths and evaluate test effectiveness from the standpoint of the number of logic paths actually executed versus the total number of possible logic paths.

3.2.4.1.3 Program Testing Translator (PTT)

This tool is described by Stucki [32]. This tool gathers and analyzes data in two general areas: (1) the syntactic profile of FORTRAN source programs showing the number of executable, nonexecutable, and comment statements, the number of CALL statements and total branches, and the number of coding standards violations; and (2) actual program performance statistics corresponding to various test data sets.

3.2.4.1.4 Software Implementation Monitor (SIMON)

SIMON is described by Corrigan [30]. It contains as basic elements the functions of compile, test, report, and edit. The compile function performs "static module analysis" of source code and compiler output to extract information such as complexity measures, test path analysis, and static resource requirements. The precompiler and postcompiler can also insert instrumentation into the code and can monitor the coding conventions used. The test function performs "dynamic module analysis," extracting measures such as paths tested, success-failure data, and dynamic resource use. The report function disseminates status reports to project management based on data collected by the other system components. Finally, the edit function allows programmers to enter source code and information concerning the developing modules, project personnel, error data, status changes, test plans and data, and system documentation. These four modules operate on a common, hierarchical data base.



3-2. Advanced Execution Analysis Approach

3.2.4.2 Use

The execution analysis technique appears to be used on a limited basis. It is described as a computer based formal testing technique which would be utilized in conjunction with other testing techniques during the system evaluation phase of a software development project. However, this technique could also be used as an informal technique during the system implementation phase.

3.2.4.3 Advantages

The two major advantages of this technique are:

a. The automatic control and monitoring of test activities.

b. An aid in enforcing programming and system development standards.

3.2.4.4 Disadvantages

The only disadvantage of this technique is the high cost of developing the tools which utilize this technique since any one of these tools is limited to a specific programming language and/or hardware.

3.2.5 Automated Network or Path Analysis

3.2.5.1 Description

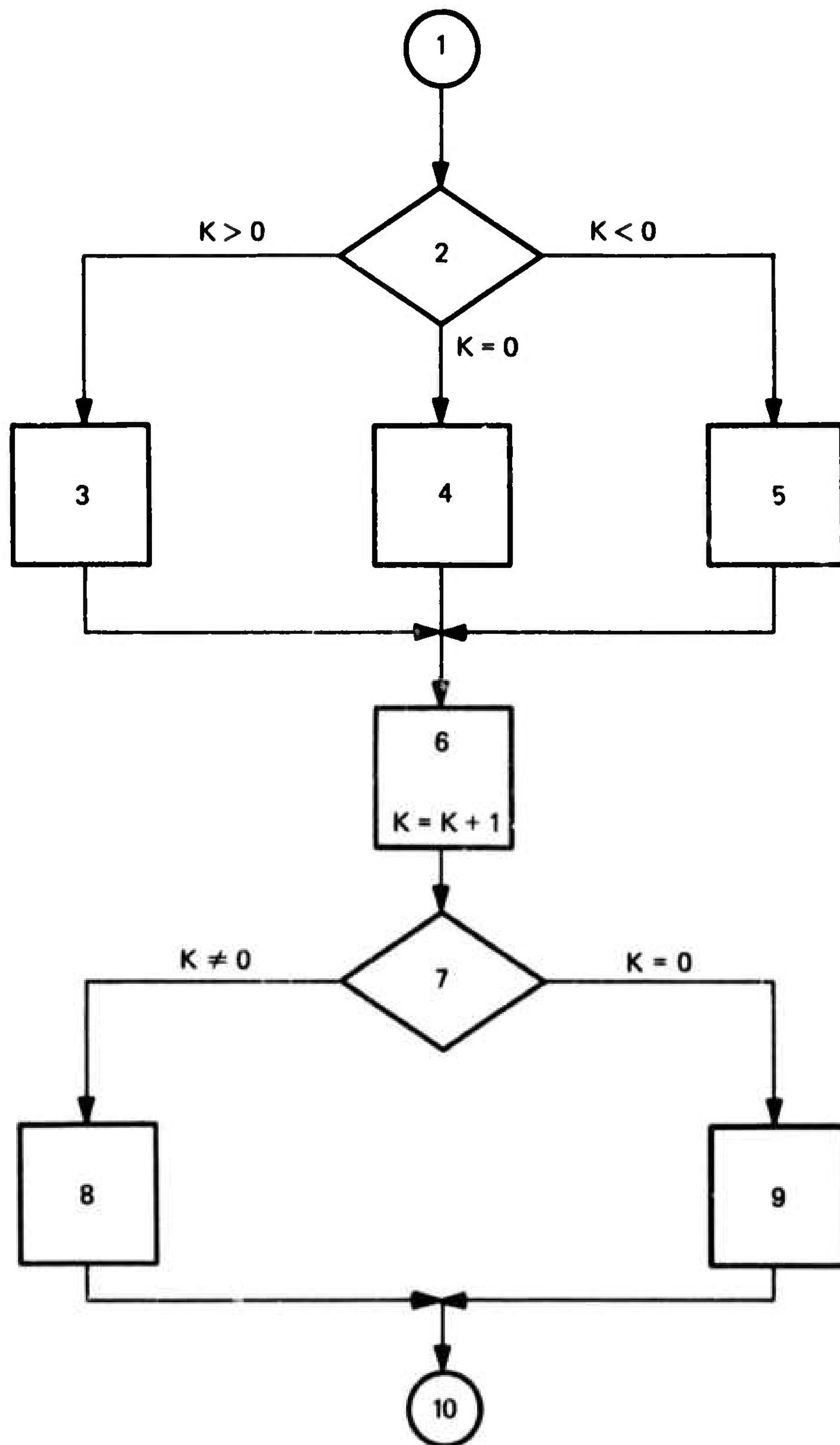
The automated network or path analysis technique defines a practical measurable means by examining source code and determining the minimum set of paths which exercise all logical branches of a program. K. W. Krause and R. W. Smith [37] describe a system that analyzes FORTRAN source code.

The following outline of this technique and Figure 3-3 were extracted from Krause and Smith [37]. They define the following terms that are used only for the description of this technique but are not used again throughout the remainder of this report.

a. Segment of code-the smallest set of consecutively executable statements to which control may be transferred during program execution.

b. Segment relationship-the relationship between two segments of code resulting from the transfer of control from the first to the second segment.

c. Impossible pair-segment relationships which cannot occur in the same path.



3-3. Example of Network or Path Analysis

d. Qualifier segment-a segment which can modify segment relationships.

e. Base path-a concatenation of segment relationships which begins at an entry point, ends at an exit point, and contains no repeated segments.

f. Loop-a concatenation of segment relationships which begins and ends at a repeated segment and contains no other repeated segments.

The automated network analysis technique uses three major processes: Source Code Analysis, Base Path and Loop Generation, and Optional Path Design.

a. Source Code Analysis-The subject module is analyzed to identify statement types and assign numbers for reference. Each segment is then analyzed to determine: (1) the segments to which each segment can transfer, (2) the segment from which each segment is accessible, (3) the type of branch expression ending a segment, (4) branch variable, (5) input variables, and (6) output variable. Once this is done, impossible pairs and qualified segments are determined.

b. Base Path and Loop Generation-In this step, the segment relationships and impossible pairs are used to generate all possible base paths and loops in a segment module. The segregation of paths and loops reduces the number of module components to a workable size.

c. Optional Path Design-The technique begins its optimization and selection process. This process begins with the selection of the base path containing the maximum number of segment relationships. This is called the characteristic path. The subset of loops which is directly or indirectly accessible from the characteristic path is identified, and the optimization process is applied to identify characteristic loops. The optimal path designed contains the maximum number of unique segment relationships in the characteristic path and its characteristic loops. Once this is determined, test data may be defined for the candidate path.

Figure 3-3 presents an example of how this technique optimizes test paths. The segments in this example are 2-3, 2-4, 2-5, 6, 7-8, and 7-9. Impossible pairs of segments are (2-3, 7-9, and 2-4, 7-9) because of the values of the parameter K. Statement 6 is a qualifier segment. There would be six paths to test in this example if all paths were to be tested: (1-2-3-6-7-8), (1-2-3-6-7-9), (1-2-4-6-7-8), (1-2-4-6-7-9), (1-2-5-6-7-8), and (1-2-5-6-7-9). Because automated analysis considers impossible pairs and qualifiers, the number of paths to be tested is four: (1-2-3-6-7-8), (1-2-4-6-7-8), (1-2-5-6-7-9), and (1-2-5-6-7-8).

3.2.5.2 Use

This technique appears to be used on a very limited basis. It is a computer based informal testing technique and could be utilized during the system implementation phase of a software development project.

3.2.5.3 Advantages

The major advantages of this technique are:

a. The automatic determination of the optimal paths to be tested for complex software consisting of an infinite number of test cases and paths.

b. An aid in enforcing programming and control structure standards.

3.2.5.4 Disadvantages

The two major disadvantages of this technique are:

a. The large number of iterations required may be excessive for certain types of software.

b. The requirements for user interaction to identify incompatible branch expressions not detected by the technique can limit its usefulness.

3.2.6 Statistical Prediction

3.2.6.1 Description

Statistical prediction is defined as the computation of a confidence factor that indicates the effectiveness of the programming and verification process by inserting errors into the software system. Mills describes this technique in his paper, "On the Statistical Validation of Computer Programs" [38]. His description of this technique is summarized below.

Through an assert, insert, test, and reject sequence, a confidence on the effectiveness of the programming and verification process can be computed. The sequence follows:

a. Assert that a given software system has no more than a selected number of "indigenous errors," e.g., $k \geq 0$.

b. Insert a selected number of "calibration errors" into the software system, e.g., $j > 0$. The insertion process consists of randomly changing or deleting source statements and could be calibrated to actual experienced software errors.

c. Test the software system until the j calibration errors are found, and record the number of indigenous errors found in a testing process, say i.

d. The confidence C is computed as:

$$C = 1 \qquad \text{if } j > k$$

$$C = \frac{j}{j + k + 1} \qquad \text{if } j \leq k$$

(This result is derived using a statistical maximum likelihood estimation technique.)

e. C represents the confidence with which we reject the assertion that $h \leq k$, where h is the unknown numbers of indigenous errors in the program. An example is shown in Figure 3-4.

The advocated method is to insert errors into the actual program on a random basis. The confidence level gained from this is a confidence in the testing process itself.

K = 0		K = 10		K = 100		K = 1000	
Calibration Errors (j)	Confidence (C)	Calibration Errors (j)	Confidence (C)	Calibration Errors (j)	Confidence (C)	Calibration Errors (j)	Confidence (C)
1	.50	1	.08	10	.09	10	.01
4	.80	4	.27	20	.17	100	.09
9	.90	9	.45	40	.28	200	.17
19	.95	19	.63	100	.50	400	.29
99	.99	99	.90	200	.66	1000	.50

Confidence that no indigenous errors remain after j calibration errors were inserted and found by testing based on assertion that no more than k indigenous errors existed and given that the testing process found no more than k indigenous errors.

3-4. Software Confidence Calculations

3.2.6.2 Use

There is no indication in the literature as to the use of this technique in a software development effort. Yelowitz [39] describes two code reading experiments where calculated errors were inserted into the source code. The first one, "Binary Search" had four seeded errors. A group consisting of nine people achieved a score of 64 percent on the seeded errors and discovered two additional errors. The group score on the total of the six errors was 46 percent. In the second experiment, "A Marketing Algorithm," four seeded errors were inserted, and one additional error was discovered. The group scores were 59 percent on the seeded errors and 61 percent on the total errors.

This technique is primarily a manual based testing technique. It must be used in conjunction with another testing technique and has the potential to be used during the system design, system implementation, and system evaluation phases of a software development project.

3.2.6.3 Advantages

Two potential advantages of this technique are:

a. An important psychological effect occurs which motivates programmers to test more rigorously to find inserted errors. The fact that a programmer knows an error exists rather than thinks one exists provides this impetus.

b. By repeating this technique several times through test phases, a statistical confidence factor can be computed which measures testing effectiveness and implies software reliability.

3.2.6.4 Disadvantages

There are several considerations that must be made prior to using this technique. They are:

a. What type of errors should be inserted?

b. How many errors should be inserted?

c. Who will insert the errors? It must be someone aware enough of the program function, yet objective enough, to "seed" realistic errors.

3.3 VALIDATION TECHNIQUES

3.3.1 Functional Testing

3.3.1.1 Description

Functional testing is defined as the execution of independent tests designed to demonstrate a specific functional capability of a program or a software system. The term functional testing is imprecise but is used because of its frequent use in the literature. Most verification and validation techniques test functions, but the tests are not designed from a functional specification viewpoint as is the case with the functional testing technique. Priority in functional testing is on specification testing rather than program testing. The primary purpose of functional testing is to validate that user requirements have been correctly programmed and, thus, its intended use is as a validation technique, but it is also used as a verification technique.

Functional testing, as described by Elmendorf [40], is a disciplined approach to testing characterized by rigorous definition of the test plan, systematic control of the test effort, and objective measurement of the test coverage. He defines five steps in the testing process to begin immediately after the system to be tested has defined its program objectives. They are:

a. The Survey establishes the intended extent of testing.

b. The Identification creates a list of functional variations eligible for testing.

c. The Appraisal ranks and subsets the functional variations so that test resources can be directed at those with highest priority.

d. The Review calculates the test coverage of the test case library.

e. The Monitor verifies that the planned test coverage was attained.

Other good descriptions of functional testing can be found in Freeman [41] and Scherr [31].

3.3.1.2 Use

This is the most frequently used validation technique. It has been used effectively as both an informal and a formal computer based testing technique. This technique and several variations were used in testing systems such as the Mission Operational Computer described in [27], the OS/360 Time Sharing Option described in [31], and all ten programming projects described in Section 4.

The functional testing technique can be used during both the system implementation phase (as a verification technique) and during the system evaluation phase (as a validation technique) of a software development project. This technique is also the one most frequently used during the maintenance of software.

3.3.1.3 Advantages

There are several advantages of this technique:

a. The testing is visible to the customer and oriented to the manner in which the customer uses the system.

b. The testing process is measurable in terms of the number of functions that have completed testing.

c. The testing of functions applies to all phases of test activities.

d. The revisions and control of the test specification is simplified.

3.3.1.4 Disadvantages

The two disadvantages of using this technique alone are:

a. All decision points of a program are not necessarily tested.

b. This technique is dependent upon a good original functional specification.

3.3.2 Design Simulation

3.3.2.1 Description

Design simulation, as used in software validation, is defined as a technique that describes a proposed system, produces a computer based "model" or simulated system, and then evaluates the effect of various system requirements and design alternatives.

In the field of software validation, Drummond [42] describes two different types of simulation that use this technique. They are trace driven simulation and the algorithm timer.

The trace driven simulation has two elements: the act of tracing applications and the act of simulating the applications on some other system. The input to the trace driven simulation model is in the form of event time and identification. In some cases, the trace driven models have as their input merely the time and type of event. In these cases, the traced input has information on what happened in the observed system rather than information on why some particular action took place. This type of simulation is often used to determine the effects of particular multiprogramming or multiprocessing approaches.

The algorithm timer is a simulation in which the concentration is on a central processing unit and its associated processor storage. The input to the algorithm timer is an assumed stream of instructions. Simpler timers require that the instruction string be "rolled out." That is, that any loops which may have been in the original program be given as a sequential string of instructions. For example, a six instruction loop which is assumed to have been executed 1000 times would be presented as a stream of 6000 instructions. This type of simulator is used for elements within the units which are extremely sensitive to the sequence of instructions and/or the location of data. The algorithm timer is most often used by the designers of products.

There are several other good descriptions of this technique referenced in the bibliography including [24], [25], and [43].

3.3.2.2 Use

This is a frequently used validation technique which was applied on three of the ten programming projects surveyed and described in Section 4. This is a computer based formal testing technique that is used most often during the system design phase but is also used throughout the entire software development cycle. In its most common use, a simulation effort is run concurrently

with design and implementation, each group using its own language, with the simulation serving primarily as a performance check on the design, but occasionally as a logic check as well.

3.3.2.3 Advantages

The major advantages of this technique are as an aid to:

a. Ensure accuracy and completeness of conceptual design.

b. Study and evaluate alternative design approaches.

c. Evaluate implementation progress and problems.

d. Determine the saturation points of various system components.

3.3.2.4 Disadvantages

The disadvantages of this technique are:

a. The tendency of the model to drift away from representing the real system.

b. The high cost of constructing and testing the simulation model.

c. The huge amount of computer time required, in some cases, to simulate a very small segment of realtime.

3.3.3 Design Validation

This technique is defined as the examination or inspection of the functional requirements and the design of a software system for the purpose of finding errors. Other terms used to describe this technique or variations of this technique include design review, project review, design inspections, walkthroughs, and inprocess reviews. This technique is similar to the design verification technique except that it is performed earlier in the software development cycle and at a system functional level. Three approaches to the design validation technique are presented below.

Army Regulation AR 18-1 [44] describes this technique in their in-process review. This review is described below under the two headings "inputs to the review" and "outputs from the review."

a. Inputs to the review:

1. Functional software documentation including

(a) General flow and process logic diagrams

(b) File layouts

(c) Edit and validity procedures

(d) Input and output formats

(e) Description of codes and notation

2. Draft of the functional user's manual

3. Personnel training plan and documentation.

b. Output from the review:

1. Interface with other systems

2. Interpretation of basic logic

3. Evaluation of whether the schedules can be met

4. Overall system design approach.

A project review is described in [45]. This review is summarized below under the headings "Objectives of the review" and "Outputs from the review."

a. Objectives of the review:

1. To review project status and communicate status to customers and higher management. Participants in the review should include project personnel (e.g., analysts, designers, programmers, and managers) and people who have not worked on the project.

2. To detect errors in the project plan or in the work accomplished thus far.

3. To determine readiness for initiating the system implementation phase.

4. To solicit recommendations for improvements in the project plan, the conceptual design, or any other aspect of the project.

b. Outputs from the review:

1. A written statement(s) by the reviewers to the project manager expressing their appraisal of the project status, likelihood of meeting objectives, specific weaknesses, and suggestions for correcting any deficiencies.

2. A written report from the project manager to upper management and/or the customer describing the results of the review, problems encountered, proposed solutions, and problems requiring upper management and/or customer assistance.

3. A common understanding of project status among the personnel present at the review.

Fagan [29] briefly describes an I_0 design inspection. This inspection takes place early in the system design phase at the functional level by inspecting the external specifications.

3.3.3.2 Use

This technique or variations of this technique have been frequently used during the system definition phase or early stages of the system design phase of a software development project. It is a manual based technique used on four of the ten programming projects described in Section 4.

3.3.3.3 Advantages

The advantages of this technique are identical to those of the design verification technique described in paragraph 3.2.3.3.

3.3.3.4 Disadvantages

The disadvantage of this technique is identical to the first disadvantage of the design verification technique described in paragraph 3.2.3.4.

3.3.4 Matrix Analysis and Problem Statement Languages

3.3.4.1 Description

Matrix Analysis and Problem Statement Languages use a forms oriented language or a formal syntax language to communicate the needs of the user to the analyst. This technique is still in the experimental stage although Merten and Teichroew [19] describe several such languages: Information Algebra, Time Automated Grid System (TAG), Accurately Defined Systems (ADS), and Problem Statement Language (PSL). These languages are primarily designed to aid in the analysis

of the functional requirements, but they also can aid the analyst in validating the functional requirements. Two approaches to the matrix analysis and problem statement language technique are presented below.

Head [46] describes TAG, an example of a matrix analysis language. TAG takes a description of the required outputs, classified by priority, format, sequence, and frequency and works backward to determine what inputs are needed and at what times. Using descriptions of inputs, outputs, and data elements, TAG can report deficiencies and inconsistencies in data specifications such as logical, time, sequencing problems, and duplicate data items.

Kosy [24] describes the "Information System Design and Optimization System" (ISDOS). This is a large research effort at the University of Michigan that contains both a Problem Statement Language (PSL) and a Problem Statement Analyzer (PSA). The PSL is used to define the system by specifying input/output requirements, data definitions, time requirements, and volume requirements. The PSA analyzes and critiques the PSL statements. The PSA program checks for proper syntax and semantic usage, logical time related elements, completeness of static interrelationships, and then generates several different reports.

3.3.4.2 Use

This technique has been used on a very limited basis according to Merten and Teichroew [19]. It appears to have potential in the future for validation by having the analyst work closely with the user in evaluating the system requirements and in facilitating the elimination and detection of logical errors by the user.

This technique is computer based, both a formal and informal technique, and could be used during the system definition and system design phases of software development.

3.3.4.3 Advantages

The advantages of this technique are best summarized by Kosy [24]:

a. Complete and unambiguous data structure definitions (e.g., transactions, tables, files, lists, queues, reports, and displays).

b. Complete and demonstrably precise interface definitions for both flow of control and flow of information, including module dependencies on the data base, dependencies on other modules, dependencies on inputs, and dependencies on parameter lists.

c. Assurance that the control logic embodied in the design will sequence the required software tasks correctly for all input combinations.

d. Assurance that each task is functionally complete and unambiguously defined.

3.3.4.4 Disadvantages

The major disadvantages of the technique are:

a. No efficient means of analyzing a problem definition given in a problem statement language.

b. The high cost of stating the requirements in a formal manner.

3.4 SUMMARY

Ten traditional testing techniques were described in this section. Of these, four computer based and two manual based techniques are classified as verification while three computer based and one manual based are classified as validation. The three techniques most frequently used are driver, test data base, and functional testing. The impact of SPT on these ten testing techniques are described in Section 5.

Section 4

STRUCTURED PROGRAMMING PROJECT SURVEY

4.1 INTRODUCTION

Ten IBM, Federal Systems Division (FSD), structured programming projects were surveyed. They covered a wide variety of applications, from the NASA Apollo Soyuz simulation to a Japanese banking system to utilities used in supporting the FAA's Enroute Air Traffic Control System. The projects surveyed were performed in Japan, California, Texas, Virginia, Maryland, and New Jersey. For each project included in the survey, there is a description of:

a. Project overview

b. Structured Programming Technology (SPT) components used

c. Test environment

d. Verification and validation techniques used

The structured programming technology components are described in Appendix B.

4.2 MITSUBISHI BANK NEW SYSTEM - JAPAN

4.2.1 Project Overview

The conversion of the Mitsubishi Bank of Japan to a new online banking system began with a special analysis and performance modeling study by FSD. Mitsubishi Bank personnel developed the new banking system. FSD provided project control services, system test plans, guidance of system test execution, performance analysis, and system tuning. New hardware and additional banking application functions were added to the new system without disturbing the existing banking system. The entire new system became operational in May 1974.

The system was developed and tested on the IBM System/370 two Models 135 and two Models 165, with interfaces to existing UNIVAC 494 processors, NCR 42 banking terminals, and IBM 3270 Video Display Terminals.

4.2.2 SPT Components

Structured programming technology components used on this project include:

a. Structured programming in Assembler Language using structured programming macros support

b. A program support library for programs in:

   1. Early stages of development

   2. System Test

   3. Operational Test.

4.2.3 Test Environment

During the system design phase, FSD prepared a comprehensive test plan which was adopted by the Mitsubishi Bank and rigorously adhered to. This plan included tests for several levels of both performance and functional testing. During preparation of the test plans, required test tools were designed and scheduled for development.

Performed primarily in an interactive mode, testing was undertaken first on the unit level and was built up through several integration levels to system testing with two full configurations. A separate test control group was set up to handle the system testing activity.

4.2.4 Verification and Validation

The verification and validation techniques used include:

a. Two test data drivers written to allow concurrent program development of different parts of the system. These drivers were used to simulate both the I/O and the critical application functions.

b. A test data base used during the implementation phase. This data base was prepared by extracting and reformatting selected data from the operational data base and system message journals.

c. Functional testing used during the system evaluation phase. These tests were prepared manually from a matrix of data base/transaction combinations.

d. Design simulation used in the preliminary feasibility study. This technique was primarily used to estimate system performance and to balance channel loadings.

e. Other verification and validation techniques, not described in Section 3, were also used. They include:

   1. A System Measurement Instrument (SMI) aided in the performance measurement of the CPU and I/O channels. This technique helped to locate bottlenecks.

2. A statistical technique was used to determine the "end of testing" during the system evaluation phase. The curve of outstanding problems for a specified time period was compared to the curve of corrected problems for the same time period. When the curve of corrected problems became parallel to the curve of outstanding problems, the system was considered ready for operational use.

3. Coding optimization techniques were used which are described by D. B. Martin [47] and Joseph H. Green [48]. These techniques were used after the results of the design simulation were evaluated.

4.3 GROUND SUPPORT SIMULATION COMPUTER FOR ASTP - HOUSTON, TEXAS

4.3.1 Project Overview

The Apollo-Soyuz Test Project (ASTP) is a combined USA/USSR flight test experiment which will join a manned Apollo spacecraft and a manned Soyuz spacecraft in earth orbit. The primary objective of this joint flight is to conduct space experiments investigating the compatibility of systems used in rendezvous, docking, and crew transfer between future USSR and USA manned spacecraft and stations.

The Ground Support Simulation Computer (GSSC) developed for Skylab will be modified to provide the flight controller training for the Apollo-Soyuz mission. The GSSC simulates the Apollo and Soyuz vehicles, experiments, and the data communications network. It generates and sends massive amounts of realistic mission radar and telemetry data to the Mission Operational Computer (MOC) for flight controller evaluation. It simulates all phases of the mission for training in nominal (normal flight) and contingency (abnormal) situations on the IBM System/360 Model 75.

4.3.2 SPT Components

Structured programming technology components used on this project include:

a. Top down programming

b. Structured programming in Assembler Language using structured programming macro support

c. A program design language

d. Program support libraries.

4.3.3 Test Environment

A formal test plan was required and periodically reviewed by the customer. An overview description of the functions to be tested was prepared. More precisely defined verification tests and expected results would have increased the effectiveness of the independent test teams during the latter stages of the implementation phase, especially in testing functions affecting several subsystems.

A Remote Test Facility developed for earlier projects allows testers to simulate the ASTP realtime operational environment in one System/360 Model 75 with no external hardware. This facility provides a jobshop environment for testing at a systems level, reducing resources required for testing and decreasing duplicate testing efforts.

Only two levels of testing were performed on GSSC due to top down structured programming: function tests during the implementation phase and function tests during the evaluation phase.

4.3.4 Verification and Validation

The verification and validation techniques used include:

a. A driver which simulates the command modules and sends "astronaut actions" to the GSSC.

b. Functional testing which is performed for nominal situations and for contingency situations.

c. Design verification of new modules, using walk-throughs, before implementation begins.

d. Design validation which is used when major modifications are required for the complex realtime system.

e. Top down testing performed with major functions being tested as they were developed, followed by the tests for subordinate functions developed later in the implementation phase.

4.4 ADVANCED CONTROL SYSTEM (ACS) -- HOUSTON, TEXAS

4.4.1 Project Overview

A processing control system for oil refineries is being developed for Imperial Oil Enterprises, Limited (IOEL) of Canada and ESSO of Belgium. ACS is a real-time system which monitors and controls the continuous and batch processes that are found in a crude oil refinery. ACS is designed to allow user interaction with the system through console groups composed of combinations of IBM 3277 (black/white) and IBM 5985 (color) display terminals. ACS is also used in planning, initiating, monitoring, controlling, and performing shutdown in the blending of gasoline and other products produced in the refinery. The system is currently in use at the IOEL Strathcona refinery, Edmundton, Canada.

The system was developed and tested on the IBM System/370 and IBM System/7.

4.4.2 SPT Components

Structured programming technology used on this project includes:

a. Structured programming in Assembler Language using structured programming macro support

b. A program design language

c. Program support libraries

d. A programming librarian.

4.4.3 Test Environment

A detailed system test plan was prepared for each refinery covering multiple computer and refinery field interfaces. A separate test team comprised of IBM and customer personnel performs the system test. Discrepancies found by the team are tracked in daily reports which reflect discrepancy activity.

Most testing is done in interactive mode using the IBM System/370 Real Time Operating System (SRTOS). Four levels of testing are performed for ACS:

a. Unit tests during the implementation phase.

b. Integration tests by the development test departments for possible subsystem impact of interdependent code during the final stages of the implemenation phase.

c. System tests of all functions during the final stages of the implementation phase.

d. System acceptance tests with a complete regression test series and customer engineer tests during the final stage of the evaluation phase.

4.4.4 Verification and Validation

The verification and validation techniques used include:

a. Drivers are used because the software and special hardware are being developed concurrently. They are used as:

   1. Hardware simulators for field elements to provide inputs and status information

   2. Hardware simulators for tank gauging and digital temperature indicators.

b. Test data bases were used.

c. Functional testing was performed with:

   1. Test narratives which list a general description of the function to be tested, the capabilities to be tested, verification method, and data base requirements.

   2. Test cases which detail the steps to test the capabilities identified in the test narrative.

d. Another verification and validation technique, not described in Section 3. Analysis is performed by a hardware monitor of I/O utilization which revealed bottlenecks in I/O processing.

4.5 EARTH RESOURCES INTERACTIVE PROCESSING SYSTEM (ERIPS) - HOUSTON, TEXAS

4.5.1 Project Overview

ERIPS analyzes remotely sensed earth resources data. The system processes uniquely formatted tape image data returned after each Skylab mission as well as processing data recorded by aircraft and satellites. Using ERIPS enables scientists to study seismic, agricultural, geological, and oceanographic information, and sources and patterns of air pollution. The system was developed and tested on an IBM System/360 Model 75 with special purpose terminals.

4.5.2 SPT Components

Structured programming technology components used on this project include:

a. Top down programming

b. Structured programming in Assembler Language using structured programming macro support

c. Program support libraries

4.5.3 Test Environment

Acceptance test specifications were prepared during the design phase at the same time as the specifications were prepared for program development. Testing is done in an interactive mode via special purpose terminals containing a conversational screen, image screen, color monitor, keyboard, and cursor control.

There are three phases of testing for ERIPS:

a. Top down testing resulting from using top down programming.

b. Inclusion of modifications and new capabilities after a regular system build. A separate verification team performs extensive testing of new modifications and some regression testing.

c. Before periodic release to the user of any version of ERIPS. The verification team performs an exhaustive test of all system functions to ensure integrity of the entire system.

4.5.4 Verification and Validation

The verification and validation techniques used include:

a. A test data base which is used during development and is controlled by the program development manager. For system testing, a separate test data base is employed. The operational data base is used with the released system for scientific analysis.

b. When function tests are being developed, the tests are set up so that all reading of the data base takes place before any updates, thus, minimizing data base problems.

c. Design validation and design verification are performed as part of the customer ERIPS interface.

d. Top down testing is performed in both an interactive and batch environment. In the past, most testing was interactive. The batch testing capability has been upgraded as part of the operational system and thus will be more heavily used in the future.

4.6 FAA NATIONAL AIR SPACE SYSTEM SUPPORT SOFTWARE (NASCOR, ACEUTE) - ATLANTIC CITY, NEW JERSEY

4.6.1 Project Overview

The Federal Aviation Administration (FAA) National Air Space (NAS) system is currently in use at 20 Air-Route Traffic Control Centers (ARTCCs) throughout the United States. Each ARTCC is responsible for controlling flights through its assigned air space which is divided into geographical sectors.

Flight plan maintenance and flight monitoring are primary NAS functions. Radar data is received by the system, processed and classified, then matched against previously received flight plans and identified.

Each center has a unique set of site-dependent adaptation data in its data base, all the relatively stable information needed by NAS. The program generating this adaptation data is called ACES.

Testing such a system requires considerable software support. NASCOR provides formatted printing services and analysis services of a NAS core dump tape. ACEUTE is a utility which analyzes adaptation data, provides lists, and generates system test tapes from an ACES system input tape. These utilities were tested and developed on an IBM System 9020 using OS/MVT.

4.6.2 SPT Components

Structured programming technology components used in this project include:

a. Top down programming

b. Structured programming using the JOVIAL language

c. Program design language

d. Program support libraries generated by the NAS support software group for use with JOVIAL and its data tables (compools)

e. Programming librarian for the development of ACEUTE.

4.6.3 Test Environment

During implementation, testing was in batch mode and a separate test team was used. Three levels of testing were performed on the NASCOR/ACEUTE utility programs:

a. Unit tests during the implementation phase.

b. Preliminary system testing of all baseline functions one week before release to the NAS System Test Department.

c. Continuous checkout while being used as system test tools by the NAS system test group during and after the evaluation phase.

4.6.4 Verification and Validation

The verification and validation techniques used include:

a. A test data base was created for NAS testing to provide a fictional center with a corresponding set of adaptation data to describe it. It was designed to incorporate a nominal set of data and all the idiosyncrasies of all ARTCCs. It is used for all levels of NAS testing at the National Aviation Facilities Experimental Center (NAFEC) in Atlantic City, New Jersey and also in unit testing NASCOR/ACEUTE. For a preliminary system test, the actual tapes of the ARTCC adaptation data are used in verifying NASCOR/ACEUTE functions.

b. The functional objectives were established and presented in a user's guide. Functional testing was performed from the user's guide.

c. The system design was verified with a design review.

d. Very recently, code verifications of NASCOR/ACEUTE.

e. Top down testing was used which resulted in fewer errors as integration difficulties were identified and solved early. Considerably fewer errors were observed in using the structured programming techniques compared to the more traditional techniques.

4.7 RESOURCE MONITORING SYSTEM FORCE STATUS - ROSSLYN, VIRGINIA

4.7.1 Project Overview

Maintenance of data coding systems that facilitates processing and exchange of military resources within and between components of the Department of Defense

(DoD) concerned with force and facility readiness is a Joint Chiefs of Staff requirement. The Resource Monitoring System (RMS) objective is to provide a single framework in which resource data may be maintained and exchanged. RMS is composed of the Force Status (FORSTAT) information system and three associated Reference File subsystems. The system was developed for a Honeywell computer.

4.7.2 SPT Components

Structured programming technology components used on this project include:

a. Structured programming in ANS COBOL

b. Program design language

c. HIPO

d. Program support libraries generated for FORSTAT for use with Honeywell COBOL

e. Programming librarians (maintenance/development/test).

4.7.3 Test Environment

The test plan lists capabilities, key tests, and schedules. The system test plan contains very detailed test sheets for new development areas. There are multiple versions of the system to handle new modifications and changes to the base system. These multiple versions must be very carefully controlled to prevent problems. Since the system is very dynamic, the development team was reorganized to provide a separate test team in time for the system test effort.

Testing is primarily in the batch mode with two levels of FORSTAT testing:

a. Unit testing during the implementation phase.

b. Functional testing during the evaluation phase.

A number of test cases are stored in a disk library with more to be developed and added in the future.

4.7.4 Verification and Validation

The verification and validation techniques used include:

a. A driver used to build the data base for testing.

b. A small subset of the operational data base used for system test due to space constraints.

c. Functional testing used to validate the capabilities described in the test plan.

d. Code verification performed by a peer on all code before testing. This is considered to be so valuable that it is a formal part of each programmer's task.

e. Design verification performed on the program design language by using structured walk-throughs.

f. Other techniques, not described in Section 3, were also employed. Manual based path analysis and matrix analysis were used in preparing the detailed system tests and charts for test results.

4.8 MULTI-OPTICAL SENSOR SIMULATION (MOSS) - WESTLAKE, CALIFORNIA

4.8.1 Project Overview

The Multi-Optical Sensor Simulation (MOSS) is an extension of the Spacetrack Augmentation Data Processing Simulation (SADPS) study performed earlier for the Space and Missile Systems Organization in Los Angeles, California. The MOSS provides a software functional simulation of optical sensors, both groundbased and aboard surveillance spacecraft. Developed concurrently was a prototype software system which simulates data processing of the optical sensor data through trajectory determination. The system was developed, tested, and demonstrated on the IBM System/360 Model 75.

4.8.2 SPT Components

Structured programming technology components used on this project include:

a. Chief programmer team concept*

b. Top down programming

c. Structured programming in FORTRAN with a preprocessor

d. Variation of HIPO* including:

*Although these are not considered part of SPT defined in Appendix B, they are included for purposes of the project survey.

1. Simulator inputs

2. "Design notes" - algorithm specifications, how to do function, where applicable

3. Output definitions

e. Program support libraries.

4.8.3 Test Environment

A system test plan was required by the customer who approved and added to what FSD presented. There was no separate test team because of the relatively small size of the project and the limited period of performance.

A preprocessor was used to add the IFTHENELSE/DOWHILE/DOUNTIL capabilities and processing without labels. It was strongly felt that structured programming without the proper tools reduced the advantages considerably. Top down programming was performed during batch job processing and testing was in batch mode wherever possible. Block time was necessary at times to ensure availability of computer resources and to collect performance statistics. There was one level of testing for MOSS but two types of tests: customer defined acceptance tests and 'additional' tests designed by FSD to exercise other options.

4.8.4 Verification and Validation

The verification and validation techniques used include:

a. Two drivers, necessary due to concurrent sensor simulation/ground software development. They were used to generate a data base until the sensor simulation was available to generate the data base and to simulate the I/O in order to run the tests.

b. Test data base used during the early stages of the implementation phase.

c. Functional acceptance and 'additional' tests performed during the evaluation phase.

d. Code verification used when a problem was encountered and for certain critical paths during the implementation phase.

e. Top down testing, the only computer based testing, performed during the implementation phase of MOSS.

4.9 TEST CONTROL COMPUTER SUPPORT SOFTWARE (TCCSS) - GAITHERSBURG, MARYLAND; ENDICOTT, NEW YORK

4.9.1 Project Overview

The Test Control Computer Support Software (TCCSS) written in Gaithersburg was part of a joint effort between System Products Division (SPD) and FSD. The objective of TCCSS was to extend the operating services to establish a computer environment in which the problem (application) program could run tests of hardware logic circuitry. The system was developed and tested on the IBM System/370 Models 145 and 155 with OS/VS2 and OS/MVT.

4.9.2 SPT Components

Structured programming technology components used in this project include:

a. Top down programming

b. Structured programming

c. Program design language

d. Program support libraries

e. Programming librarian

A problem was encountered in reconciling an interactive library system and a librarian where the relationship between the source modules and their generated code is confusing.

4.9.3 Test Environment

The TCCSS test plan described the functions to be tested. The delivery schedules were revised to accommodate structured programming milestones.

a. The completion of the program design (i.e., PDL was used) replaced the traditional unit test milestone.

b. Top down programming supplanted integration of modules. Testing was performed as the system developed without a separate test team. Acceptance testing was carried out by the SPD "customer" in Endicott, New York.

   Testing was interactive as it is a terminal-driven system and compatibility between OS/VS2 and OS/MVT was necessary. Most of the testing was performed using block time in order to get the resources required for "top down" testing and integration.

Two levels of testing occurred for TCCSS:

1. Top down testing during the implementation phase. Some unit testing due to an access method problem.

2. Acceptance testing by the customer during the evaluation phase.

4.9.4 Verification and Validation

The verification and validation techniques used include:

a. Small drivers developed due to system software problems.

b. Test data bases used during the early stages of the implementation phase.

c. Functional testing used during both the implemenation and evaluation phases.

d. Design verification performed by using structured walk-throughs.

e. Top down testing.

4.10 TRIDENT AN/BQQ-6 SONAR SOFTWARE - MANASSAS, VIRGINIA

4.10.1 Project Overview

The Navy's TRIDENT AN/BQQ-6 Submarine has advanced technology sonar software and hardware which are being developed concurrently in Manassas. Though earlier software was developed traditionally, all new software will use structured programming technology. The sonar software is being developed and tested on the IBM System/370 and AN/UYK-7 computers.

4.10.2 SPT Components

Structured programming technology components used on the project include:

a. Top down programming

b. Structured programming with a preprocessor

c. Program support libraries for programs in development, test and integration, and a master library for programs ready for system test.

4.10.3 Test Environment

In the AN/BQQ-6 environment, prior to hardware/software integration, there are two software testing paths which coincide with the concurrent hardware and software development:

a. The traditional hardware testing cycle utilizing unit and subsystem test software followed by use of functional elements of the operational software. This latter testing confirms the validity of the hardware/software interfaces and timing.

b. A second path following the development testing milestones used for structured programming management with a multi-library concept:

   1. Development Testing - Utilizes development library with testing via the language system facilities, to perform initial functional testing.

   2. Test and Integration - Utilizes integration library and hardware simulator for extensive functional testing and initial software system concurrency testing.

   3. Inclusion in Master Library - Library is under Configuration Management Control, and testing of the programs via the hardware simulator is accomplished by an independent test group, utilizing formal test plans and test procedures. The objective is functional testing against requirements and full software concurrency testing.

The independent test group used Navy approved system requirements documents, the approved Computer Program Performance Specifications (CPPS) and the Computer Program Design Specifications (CPDS) to prepare the system software tests.

Testing is performed in the batch mode during the early stages of the implementation phase and in an interactive mode during later stages. There are four levels of testing for AN/BQQ-6: implementation test, independent software integration, hardware/software integration test, and system design certification test which is the final formal customer acceptance.

4.10.4 Verification and Validation

The verification and validation techniques used include:

a. A hardware simulator developed to drive the software.

b. A test data base or compool.

c. Functional testing performed from matrices of system functions.

d. Three manual techniques employed -- design validation, design verification, and code verification.

e. Design simulation used for performance evaluation.

f. Top down testing. It enabled functions to be prepared in level 1, levels 2, and partial package increments to coincide with the preliminary hardware development.

4.11 ENERGY MANAGEMENT SYSTEM (EMS) - HOUSTON, TEXAS

4.11.1 Project Overview

An energy management processing control system for electric power plants is being developed in Houston. It is currently being used in Beaumont, Texas, and Baton Rouge, Louisiana. The system is being developed and tested on IBM System/370 and System/7 computers with special purpose consoles which contain a conversational screen, keyboard, and color schematic screen.

4.11.2 SPT Components

Structured programming technology components used on this project include:

a. Structured programming in Assembler Language using structured programming macro support

b. Program support libraries

c. Programming librarian.

4.11.3 Test Environment

The test plan provides for seven different test cells (hardware/software configurations) for the IBM System/7 process control computers and four test cells for the IBM System/370.

Three test teams, one for the IBM System/370, and two for the System/7 computers, verify each system function using a test script. These test scripts are grouped into a test case covering the functions to be tested for a given test cell. The results are tracked automatically. All testing is performed interactively at the console by:

a. Generating the system (SYSGEN), linking in cards defining the system, building the data base, and then starting the System/370 Real Time Operating System (SRTOS).

b. Running the test case for a given test cell by running as many test scripts for the test case as possible in the time allotted, noting results and discrepancies carefully. Discrepancies are tracked in daily reports.

EMS uses four levels of testing. They are:

a. Unit testing within the system environment, monitoring the changes as they are included during the implementation phase.

b. Integration testing in the IBM System/7 area.

c. System functional/performance testing during the latter stages of the implementation phase.

d. Acceptance testing with an IBM test team from General Systems Division (System/7) and Data Processing Division (System/370) during the evaluation phase.

4.11.4 Verification and Validation

The verification and validation techniques used include:

a. A test data base generated to provide sample generator outputs and transformer analog input points. Operational data bases for EMS will be prepared by each power company using it to adapt the system to its own equipment configuration and operating environment.

b. Functional testing where functions were specified and tested for each of the different hardware and software configurations laid out for the System/370 and System/7 computers.

c. Design verification used on a limited basis.

d. Design simulation and performance monitoring used on a limited basis.

Section 5

STRUCTURED PROGRAMMING TECHNOLOGY (SPT) IMPACT ON VERIFICATION AND VALIDATION

5.1 INTRODUCTION

Most of the discussion presented in this section is based on analysis and experience. Structured programming technology is still in its infancy, and thus, conclusive statistical data is unavailable at this time. The following is an initial attempt to relate structured programming experience obtained from the literature and project surveys to software verification and validation.

Structured programming technology, described in Appendix B, has provided the impetus for the increased utilization of two verification techniques: (1) top down programming and its associated computer based testing and (2) manual based code verification. Neither technique is new. Top down programming is a component of SPT, while code verification is usually associated with structured programming because of the following reason. Structured programming has made programs easier to read. The reader can read top down structured programs in a sequential systematic way, in order to follow the requirements being implemented in the program. The reader of a top down structured program doesn't have the problems of determining which branches or jumps to look for first or how to keep track of various branches or jumps that he has uncovered during his reading process. These two techniques are further discussed in subsection 5.2.

Structured programming technology has also had an effect on some of the previously discussed verification and validation techniques. The effect on these techniques is discussed in subsection 5.3.

5.2 TECHNIQUES ASSOCIATED WITH SPT

5.2.1 Top Down Programming

This technique is defined as performing in hierarchical sequence a detailed design, code, integration, and test as concurrent operations. A detailed description of this technique is presented in Appendix B.

Two authors' viewpoints on top down programming and the resultant top down testing technique are presented below.

Dijkstra [49] discusses the technique used in testing the THE multiprogramming system. He suggests a system in a hierarchy of layers, each layer corresponding to an abstraction of the same layer below. In this sense, each layer is a new machine where the software of layer i transforms machine A(i) into A(i+1). Each layer's instructions (operations) are interpreted by the machine

on the next lower level. Each instruction can be viewed as an abstraction of the operations which executes it on the lower level. As each level of the system is completed, it is tested to force that level into "all relevant states" (one may argue whether all relevant states are attained). Dijkstra contends that by not viewing the system as a "black box" but rather a layered structure, the set of relevant test cases is reduced to a manageable number. Consequently, if each level has been independently and exhaustively tested, when the system is complete, it can be assumed to be correct.

Mills [50] describes "systems of code" by generating a sequence of intermediate systems of code and functional subspecifications so that at every step, each system can be validated to be correct (i.e., logically equivalent to its predecessor system). He describes the initial system as the functional specification for the program, each intermediate system includes the code of its predecessor, and the final system is the code of the program.

Mills concludes that the problem of proving the correctness of any expansion of a functional subspecification is reduced to proving the correctness of a segment (i.e., a page of code with one entry at the top and one exit at the bottom) in which, possibly, various named subspecifications exist. The verification of a given segment requires a proof that the segment subspecification is met by the code and named subspecifications. He further concludes that the named subspecifications will be subsequently verified, possibly in terms of even more detailed subspecifications, until segments with nothing but code are reached and verified.

5.2.2 Use

The literature survey provided very few references to software development efforts that used this technique. Six of the ten programming projects, described in Section 4, used this computer based informal testing technique. This technique is used exclusively during the system implementation phase of a software development project.

5.2.3 Advantages

The advantages of this technique are:

a. The set of relevant test cases is reduced to a manageable number.

b. The effort required to produce drivers that pass data to modules for testing is significantly decreased.

c. The software product is evolved to maintain the characteristic of always being operable.

d. The quality of a program produced using this approach is increased, as reflected in fewer errors in the coding process.

e. The early resolution of module interfaces.

f. The computer testing time requirements are spread more evenly over the development cycle.

5.2.4 Disadvantages

The disadvantages of this technique are:

a. The management problem of planning and controlling the coding and testing processes by not allowing developers to perform a large number of independent activities in parallel.

b. Some modules cannot be completely tested until subordinate modules are developed and are available.

5.2.2 Code Verification

5.2.2.1 Description

This technique is defined as the examination or inspection of a computer program for the main purposes of uncovering logic errors in the program design and finding coding errors in the source code. Other terms used to describe this technique or variations of this technique include code reading, program reading, code review, design inspection, desk checking, walk-throughs, and structured walk-throughs. Four approaches to the code verification techniques are presented below.

Mills [12] describes a technique for inspecting segment structured programs. This technique will be summarized under two headings: "reading instructions" and "modes of inspections."

a. Reading instructions

   1. The reader should identify every basic control figure that occurs in a program.

   2. The reader should examine the proposition for each basic control figure for its truth or falsity.

   3. The reader should review the test patterns for the extent to which they exercise text and for the results compared with requirements.

b. Mode of inspections

   1. The author of the program should begin the inspection with the designing and writing of the segment, so that program, proof, and test patterns are developed jointly rather than in sequence.

2. Programmers, already involved in writing other segments of the program, should read ancestor segments and test results to understand the environment for the new segments. People assigning new segments to others should read these segments and tests results, when completed, to verify their correctness. This mode of reading should be a normal part of any program development.

3. Programmers, outside the development team, should inspect the program after the program is complete or concurrently with the development. The advantage of this mode over (2) is the fewer biases of the inspection team. A disadvantage of this mode over (1) is that the team may not understand the context of a segment as well.

A structured walk-through is a generic name given to a set of techniques (i.e., design validation, design verification, and code verification), each with different objectives and each occurring at different times in the software development cycle. Structured walk-throughs are described in [45] and [51]. Procedures, associated with code verification, were extracted from these sources and are presented under the headings of "characteristics of a walk-through," "items to be reviewed," and "output from a walk-through."

a. Characteristics of a walk-through

1. It is arranged and scheduled by the developer (reviewee) of the work product being reviewed.

2. Management does not attend the walk-through and it is not used as a basis for employee evaluation.

3. Prior to the walk-through, the participants (reviewers) are given the review material and are expected to be familiar with it.

4. The walk-through is structured so that all attendees know what is to be accomplished and what role they are to play.

5. All technical members of the project team, from most senior to most junior, have their work reviewed.

6. A typical walk-through will include four to six people and will last for a prespecified time usually one or two hours.

b. Items to be reviewed

1. Program specifications

2. Test preparations

3. Uncompiled source listings

4. Test results.

c. Output from walk-through

1. A designated recording secretary records all the errors, discrepancies, and inconsistencies that are uncovered during the walk-through.

2. The recording secretary generates an official action list for the reviewee which is also used as a communication vehicle with the reviewers.

Fagan [29] presents an I_2 design inspection that is used as a code verification technique after the first clean compilation of a program. His I_2 design inspection is summarized below under the three headings: "inspection team," "outline of the inspection procedure," and "examples of what to examine when looking for errors." The inspection team and the outline of the inspection procedure is similar to the I_1 design verification technique described in paragraph 3.2.3.1. There are substantial differences, however, so the entire description is given for both techniques.

a. Inspection Team

1. Moderator-The key person in successful inspection. He need not be a technical expert, but he must manage the inspection team and offer leadership. He must use personal sensitivity, tact, and drive in balanced measures. His use of the strengths of team members should produce a synergistic effect larger than their number. He is The Coach.

2. Designer-The programmer responsible for producing the program design.

3. Coder/Implementor-The programmer responsible for coding the design.

4. Tester-The programmer responsible for writing and/or executing test cases or otherwise testing the product of the designer and coder.

In the event that the coder of a piece of code also designed it, he will function in the designer role, and a programmer from some related or similar program will perform the role of coder.

In the event that the same person will design, code, and test the product code, the coder role should be filled as described above. Another programmer, preferably with testing experience, should fill the role of tester.

b. Outline of the inspection procedure

1. Preparation (individual)-Participants, using the source code listings, try to understand the implementation of the design.

2. Inpsection (whole team)-The coder describes how he implemented the design. Every piece of logic is covered at least once, and every branch is taken at least once. The primary objective is to find errors and not to redesign, evaluate alternate design solutions, or find solutions to errors.

3. Follow-up-It is imperative that every issue, concern, and error be entirely resolved at this level. Errors can be 10 to 100 times more expensive to fix if found later in the process (programmer time only, machine time not included).

c. Examples of what to examine when looking for errors.

1. Test branch

(a) Is correct condition tested (If X = ON vs. If X = OFF)?

(b) Is the correct condition used for test (If X = ON vs. If Y = ON)?

(c) Are null THENs/ELSEs included as appropriate?

(d) Is each branch target correct?

(e) Is the most frequently exercised test leg the THEN clause?

2. For each interconnection call to either a macro, or another module:

(a) Are all required parameters passed correctly?

(b) If register parameters are used, is the correct register number specified?

(c) If interconnection is a macro, does the inline expansion contain all required code?

(d) Are there register or storage conflicts between macro and calling modules?

(e) If the interconnection returns, do all returned parameters get processed correctly?

Another approach to code verification is a code review checklist used in the training of new programmers at FSD of IBM [52]. The checklist is subdivided into three categories: readability, program logic, and mechanics.

a. Readability

1. Is the program easily readable and understandable?

2. Are the standard indentation rules followed?

3. Are names contextually understandable and useful?

4. Are there any additional comments?

5. Are additional comments needed?

6. Is the program properly segmented?

7. Are any explicit branch statements used? If yes, are they justified?

8. Do any "DO Groups", begin blocks, or "IF Groups" extend over one segment?

b. Program Logic

1. Is the logical approach valid?

2. Is the logical approach unnecessarily complex or confusing?

3. Is invalid input considered and handled?

4. Are all assumptions clearly stated?

c. Mechanics

1. Are there any syntax errors?

2. Is there any possible misuse or poor construction of instructions?

3. Are the conventions of writing instructions followed? That is, one data item per line, no defaults taken.

4. Are the standard figures implemented properly (e.g., case)?

5.2.2.2 Use

The code verification technique is being used for several other purposes in addition to its main purpose of finding errors. They include:

a. Teaching aid-Junior programmers or programmers unacquainted with the the system can gain experience in good programming practices or a better understanding of the evolving system.

b. Management review aid-Management can gain an insight into the progress and problems that a programmer might be experiencing.

The literature indicates that the major use of the code verification technique has been for the purpose of find errors. Four of the ten programming projects which are described in Section 4 used this technique for this purpose. Fagan [29] describes the use of this technique on a wide basis with excellent results. Several experiments on code verification are also described in the literature. They include: Yelowitz [39], Jelinski and Moranda [53], and Corrigan [30]. The first two reports describe code reading experiments with the objectives of developing a basis for establishing the economics of code reading as a means of testing. Corrigan [30] describes the use of code reading on the SIMON precompiler project. Although quantitative data from the SIMON project is not presented, the code readers were convinced of the utility and value of public code reading.

This is a manual based technique used primarily during the system implementation phase of a software development project.

5.2.2.3 Advantages

The advantages of this technique are:

a. The error rework cost decreases due to the early detection of errors.

b. The quality of the software product is improved because in many cases errors are found that would not be found during subsequent computer based testing.

c. An increase in programmer productivity is achieved due to a positive psychological effect that code verification has on programmers.

d. The use of the code verification technique as both a teaching and management review aid.

e. The experience gained from code verification can be a valuable asset during the computer based testing activities.

5.2.2.4 Disadvantages

The disadvantages of this technique are:

a. The lack of enthusiasm by the people performing a code verification. Code verifications are considered by some people to be tedious and boring and one of the least desirable tasks performed by a programmer.

b. The use of the code verification technique in some application environments may not be cost effective.

5.2.2.5 SPT Impact on Code Verification

SPT has directly affected code verification in several ways. They include:

a. A Programming Support Library (PSL) makes developing code more visible and accessible by providing hard copy listings that correspond to the most current version of the system.

b. Top down programming (e.g., segmented code) and structured programming (e.g., limited set of control figures and indentation rules) make code more readable and also make it possible to read someone elses code with greater comprehension.

c. A Program Design Language (PDL) provides a more standard means of communication between designer and programmer, and thus, assists the code verifier in understanding both the design and its implementation. The concepts of structured programming are applied to a PDL in the form of basic control structured for logic flow and indentation. Top down programming is implemented by specifying in PDL the top level portion of the program and evolving in the PDL into succeeding levels of detail. These two attributes of a PDL provide a more natural relationship to programming languages than traditional methods, thereby enhancing the code verification by comparing the similar PDL statements with the source code.

Prior to the advent of structured programming, coding and segmentation standards were usually not enforced. The lack of enforcement of these standards made code verification difficult, time consuming, and almost impractical. Because of the SPT benefits described previously and the additional benefit of being able to read a program in a sequential, systematic way, code verification is evolving as a practical and cost effective verification technique.

5.3 SPT IMPACT ON TECHNIQUES DESCRIBED IN SECTION 3.

There is no discernible effect of SPT on three of the techniques described in Section 3. The three techniques are statistical prediction, design validation, and matrix analysis/problem statement languages. A fourth technique, test data bases, is affected only by the Programming Support Library (PSL) component of SPT. A PSL, as described in Appendix B, provides capabilities for storing and listing test data files. The other techniques described in Section 3 are more significantly affected by SPT. These effects are described in the following paragraphs.

5.3.1 Drivers

SPT has a major impact on drivers. The major reasons for this positive impact is due to using the top down programming technique which is described in Appendix B. Several key points of Top down programming were abstracted from that discussion and are presented below to aid in discussing the impact of SPT on drivers. In the traditional software development project, the lowest level

processing programs are coded first, unit tested, and made available for integration. Superfluous code in the form of drivers is needed to perform the unit testing and lower levels of integration testing. In top down programming, coding and testing are performed "top down" in execution sequence. This programming technique minimizes the need for drivers, but does require the creation of stubs for subordinate segments of code which are to be replaced eventually with running code. These stubs, which will eventually be discarded, may contain a "no operation" or possibly an output statement to show that control has been received. Top down programming does not completely eliminate the requirements for drivers since it might be difficult for a segment to invoke certain "error" conditions or "unlikely to occur" situations in a subordinate segment. However, the effort normally expended in writing stubs and a few necessary drivers is considerably less than that required to produce all the drivers required for a traditional "bottom up" software development effort. These stubs are also easier to write than drivers, since they are operating much closer to the decision logic which they are to affect.

The PSL component of SPT also impacts this verification technique. The PSL, described in Volumes V and VI of the Structured Programming Series, will automatically generate program stubs for segments of source code which are identified in another segment of source code being added to the PSL and are not currently stored in the PSL (i.e., have not yet been coded). This capability minimizes the effort required to write the stubs that replace the traditional drivers.

5.3.2 Design Verification

The PDL component of SPT directly affects this technique. PDL is defined and briefly described in Appendix B and discussed in detail in Volume VIII of the Structured Programming Series. A program design language is intended as a:

a. Vehicle to translate functional specification into program design.

b. Replacement for design logic flowcharts.

c. Means for communication between technical and nontechnical personnel, designers, and developers.

The third intent, "means for communcating," enhances the reviewer's ability to verify the program design because PDL's English-like and the semantical/syntactical conventions minimize the ambiguities.

PDL has another impact on this technique. A formalized PDL which has syntax and semantic rules could be analyzed by a computer program. This program could automatically detect errors in the design and assist in program design verification. Errors such as unused data items, data received too late for use, and inconsistent use of data could be automatically identified.

The PSL component of SPT indirectly affects this technique. A PSL makes the program design more visible and accessible by providing the capabilities of storing and listing the current version of the program design language statements.

The other two components of SPT, structured programming and top down programming, provide minimal impact on this technique through the use of the concepts in the program design language.

5.3.3 Execution Analysis

The structured programming and PSL components of SPT indirectly affect this verification technique. Two functions performed by the tools using this technique, that are described in subsection 3.2.4, are: (1) monitoring coding conventions, and (2) identifying data paths and measuring data paths tested.

The PSL assists in the monitoring of coding conventions. These monitoring activities include:

a. Flagging (i.e., identifying on a listing) of all explicit branches (e.g., GOTO statements).

b. Flagging program language source units that exceed a maximum size to be defined by the user.

c. Flagging any lines of program source code that contain more than one source statement (e.g., a line of code is the equivalent of a single punched card).

The structured programming component of SPT and more specifically the limited number of control structures (i.e., four control structures are described in Volume I of the Structured Programming Series) enhance the identification of data paths and measurement of data paths tested. Baker's [26] paper, on the experience gained from the New York Times project, stated that "identification of paths to be tested was greatly facilitated by the use of only those formalized control structures permitted by our structured programming conventions."

5.3.4 Automated Network or Path Analysis

The structured programming and top down programming components of SPT indirectly affect this technique. In subsection 3.2.5, the technique description is subdivided into three processees: (1) source code analysis, (2) path and loop generation, and (3) optimal path design. SPT impacts this technique in the first process. Source code analysis includes determining:

a. The number and types of segments (i.e., the smallest set of consecutively executable statements to which control may be transferred during program execution)

b. The segments to which each segment can transfer

c. The segment from which each segment is accessible

d. The type of branch expression ending a segment

e. Branch variables

f. Input variables

g. Output variables.

When using top down structured programming, the small limited number of control structures and the elimination of GOTO statements decrease the variability of segments and control between segments and thus facilitate the source code analysis process. The limited number of control structures significantly reduces the complexity of path analysis.

5.3.5 Functional Testing

Top down programming and the PSL components of SPT indirectly affect this technique. Functional testing, described in subsection 3.3.1, has the primary purpose of validating that user requirements have been correctly programmed. The major characteristics of the technique are a rigorous definition of the test plan, systematic control of the test effort, and an objective measurement of the test coverage.

Top down programming, according to Mills [50], assists in the validation of functional requirements by generating a sequence of intermediate "systems of code" and functional subspecifications. At every step of the system development effort, each system can be validated to be correct (i.e., logically equivalent to its predecessor system). The "system of code" at each level of development can be correlated and validated to the functional specifications.

The PSL facilitates the systematic control of the test effort, the second major characteristic of functional testing. The PSL provides capabilities to store and list test data. It also provides capabilities to collect and report statistics on test activities during the evaluation phase of a software development project.

5.3.6 Design Simulation

The top down programming of SPT indirectly affects this technique. An essential component of top down programming is the creation of stubs as the program evolves top down from a tree structure of segments. These stubs provide the opportunity to simulate alternative approaches to unstable or long range requirements or to simulate and evaluate system performance requirements. Using the stubs for simulating function can assist in the earlier partial implementation of requirements in complex unstable software applications. If the system is evolved top down, it is always operable and able to satisfy a subset of the system requirements. As the requirements become firm, the stubs can be replaced with operational code. Graham, Clancy Jr., and DeVaney [54] describe this approach in evaluating a software design. They describe a Design and Evaluation System. Their design evaluation approach is based on the premises that the system being evaluated must be the system which is being implemented, and the evaluation results must be available prior to the entire system being operational.

Appendix A

SOFTWARE DEVELOPMENT CYCLE

A.1 INTRODUCTION

The software development cycle, as used throughout this report and shown in Figure A-1, consists of four phases (2 through 5): definition, design, implementation, and evaluation. Each of these four phases, which are similar to the project development life cycle described in the DoD Manual 4120.17-M, is defined in the following paragraphs. The deviations from this project development life cycle and the one used throughout this report are the use of the term "implementation phase" in place of the term "programming phase" and slight modifications to the functions performed during the design, implementation, and evaluation phases.

A.2 DEVELOPMENT PHASES

A.2.1 System Definition

Software systems are generally defined by a set of system requirements. Naturally, some systems definitions are much more complete, precise, and thorough than others. This is due to many factors such as complexity and experience. Usually the collection of requirements is performed by a small team whose work product is a document containing an organized statement of requirements. The requirements document is the baseline for further development and subsequent testing.

There are two major types of requirements: functional and performance. Functional requirements describe the functions the system must perform (e.g., the system must accept 10 types of input transactions and process these against an online data base). Performance requirements specify the time and space constraints which must be met (e.g., the system must be capable of accepting as input 1000 transactions every minute for a 10-hour period and it must be capable of processing input transactions at a rate of 30,000 per hour for a 20-hour period).

Requirements are generally expressed in narrative, although other vehicles for expression which offer advantages of computability are available for use.

A.2.2 System Design

Following the definition phase, a design phase is generally necessary. If the project is relatively simple or very small, a separate formal design phase may not be required.

System design is concerned with all phases of the system: hardware, software, and internal and external user procedures.

Initiation	Development			Evaluation	Operation	
Initiation	Definition	Design	Implementation	System Test, Installation	Maintenance	Revised Operation
1	2	3	4	5	6	7

Figure A-1. Project Development Cycle

The activity during this period is directed toward producing a design which when implemented will result in a system that satisfies all functional and performance requirements. The products of this phase are system specfications, system test plans, cost estimates, and implementation plans.

A.2.3 System Implementation

System implementation includes the expansion of system specifications to enable the detailed design, build, and test of system software. The software architecture and functional specifications developed during either the definition or design phase are expanded, coded, and executed on real or simulated hardware.

A.2.4 System Evaluation

After the system is built, it is tested for conformity to system specifications, requirements, and usability. The ultimate result of this phase is the validation of the system by the personnel who requested its development.

A.2.5 Relationship of Phases

The previous discussion is not intended to convey the impression that each phase of the system development is an independent activity. The reality is that they are not independent, discrete sequential processes. There are many factors causing their overlap, including errors, omissions, changes, and funding profiles.

Appendix B

STRUCTURED PROGRAMMING TECHNOLOGY FUNDAMENTALS

B.1 INTRODUCTION

Structured programming practice has provided the impetus for an improved programming development technology. One of the first implementations utilized:

- o Top Down Structured Programming
- o Programming Support Libraries
- o Chief Programmer Team Operations

Top down structured programming includes the two interrelated concepts of structured programming and top down programming. Structured programming has a firm theoretical base in the structure theorem. A programming support library is a useful tool that serves as a repository for project data. A chief programmer team is an organizational approach for the development of software.

Each of these techniques can be utilized independently of the others, but there are dependence relationships. For example, the chief programmer team technique calls for the reading of the program source code. Program source code is generally quite difficult to read if it is not well organized and structured.

A second dependency can exist between top down programming and program support libraries. The nesting of small segments of code within other such segments as required by top down programming is best accomplished by compiler directives (e.g., INCLUDE or COPY). This implies the existence of a programming support library.

Shortly after the introduction of these techniques for production programming, the idea was introduced of expressing design in pidgin English using the ideas of top down structured programming. This technique was called program design language (PDL).

These technologies themselves are to an extent in a refinement stage; while the principles have been demonstrated, the details of implementation are still being improved. Additional techniques are being developed and associated with structured programming technology. For example, Hierarchy Input Process Output (HIPO) is a technique originally conceived as a program documentation tool. Later it was utilized in the expression of program specifications. HIPO was studied in Task 4.1.8 as a candidate for specification and/or design expression (refer to "Program Design Study," Volume VIII of the Structured Programming Series) and used in Task 4.1.6 (refer to "Programming Support Library Program Specifications," Volume VI of the Structured Programming Series).

The term "Structured Programming Technology" will be used to collectively reference the following:

- o Top Down Structured Programming
- o Programming Support Library
- o Program Design Language.

B.2 TOP DOWN STRUCTURED PROGRAMMING

Structured programming technology has its origins in structured programming, a method of writing programs based on a mathematically proven theorem. The application of this theorem results in simpler, more maintainable programs because it reduces the number of logic structures which programmers use for writing programs. A separate technique, which has become an integral part of this technology, is top down programming. This is a program development technique which results in a high degree of segmentation and permits the detailed design, production, and integration of the source program code to proceed in parallel.

The term top down structured programming is used to refer to that part of the technology which consists of the integrated use of both techniques.

B.2.1 Structured Programming

Structured programming (SP) is based on the mathematically proven Structure Theorem, due to the original form by Bohm and Jacopini [55, 56] which states that any proper program (e.g., a program with one entry and one exit) is equivalent to a program that contains as logic structures only:

- o Sequence of two or more operations
- o Conditional branch to one of two operations and return (IF a THEN b ELSE c)
- o Repetition of an operation (DO WHILE p).

Each of the three figures itself represents a proper program. A large and complex program may then be developed by the appropriate sequencing and nesting of these three basic figures. The logic flow of such a program always proceeds from the beginning to the end without arbitrary branching. When only these structures are used in the programming, there are no unconditional branches or statement labels to which to branch.

Structured programming reduces the arrangement of the program logic to a process similar to that found in engineering where logic circuits are constructed from a basic set of figures. As such, it represents a standard based on a solid theoretical foundation. It does not require ad hoc justification case by case in actual practice.

Several other practices are included as a supporting part of the technique. For example, strict attention is paid to the indentation of the logic structures on the printed page so that logical relationships in the coding correspond to physical position on the listing. Thus, a pictorial representation of the logic is gained from the indentation. Another practice is that of segmenting code into reasonable amounts of logic that are each easily understood. Each segment of code (whose internal operations may be any combination of the basic logic structures) must itself represent one of the basic logic structures. Thus, each code segment becomes a logical entity to be analyzed, coded, and read at one time.

Two extensions to the three basic logic structures (DOUNTIL and CASE) have been defined, to improve readability of source code. These do not affect the spirit of structured programming, and in some cases may result in more efficient use of computer time and storage. DOUNTIL provides an alternate form of looping structure. It differs from DOWHILE in that the condition is tested after the operation rather than before. CASE is a multibranch, multijoin control structure used to express the processing of one of many possible cases.

B.2.2 Use of Subroutines

While the use of the control logic figures eliminates the necessity of writing any explicit branch or GOTO statements (except for simulation purposes), it is not intended to preclude the use of CALL/RETURN logic to invoke subroutines. In fact, such subroutine linkages are an essential feature of top down structured programming and their use is encouraged.

B.2.3 Top Down Programming

Traditional software development has evolved as a bottom up procedure where the lowest level of processing programs are coded first, unit tested, and made ready for integration. Superfluous code in the form of driver programs is needed to perform the unit testing and lower levels of integration testing. In addition, internally formatted data has to be prepared manually and results must be checked manually. Driver and data preparation plus checking of results can easily equal the effort expended in preparation of the deliverable programs. Data definitions and interfaces tend to be simultaneously defined by more than one person and often are inconsistent. During integration, definition problems are recognized.

Integration is delayed while the data definitions and interfaces are defined consistently and the processing programs are reworked (and unit tested again)

to accommodate the changes. It is often difficult to isolate a problem during the traditional integration cycle because of the large number of possible sources of error. Management control often is ineffective during much of the traditional development cycle because even though the units are visible, there is no certainty they are correct in terms of interfaces with other such units until integration test. Top down programming is patterned after the natural approach to system design and requires that programming proceed from developing the control architecture (interface) statements and initial data definitions downward to developing and integrating the functional units. Top down programming is an ordering of program development which allows for continual integration of program parts as they are developed and provides for interfaces prior to the parts being developed. At each stage, the code already tested drives the new code, and only external data is required.

In top down programming, the program is organized into a tree structure of segments. The top segment contains the highest level of control logic and decisions within the program, and either passes control to next level segments, or identifies next level segments for in-line inclusions. The next level segments are called stubs and those which are to be replaced eventually with running code may contain a "no operation" instruction or possibly a display statement to the effect that control had been received. While it is recognized that such code as with drivers is also eventually discarded, the effort involved in writing such statements is less than that required to produce and pass data to a module for unit testing. The process of replacement of successively lower level stubs with processing code continues for as many levels as are required until all functions within a program are defined in executable code.

Many system interfaces occur through the data base definition in addition to calling sequence parameters within individual programs. Top down programming requires that sufficient data base definition statements be coded and that data records be generated before exercising any segment which references them. Ideally, this leads to a single set of definitions serving all programs in a given application.

This approach provides the ability to evolve the product in a manner that maintains the characteristics of always being operable, extremely modular, and always available for successive levels of testing that accompany the corresponding levels of implementation. The quality of a program produced using this approach is increased, as reflected in fewer errors in the coding process. The act of structuring the logic calls for more forethought, and the uniformity and single entry, single exit attribute of the structured code itself contribute to the reduction in errors.

Because of the segmented nature of top down programming, the resulting program is extremely modular in function and logic structure, minimizing the effect of requirement changes on already-developed code.

Conceptually, top down programming proceeds from a single starting point, while conventional implementation proceeds from as many starting points as modules in the design. The single starting point does not imply that the implementation must proceed down the hierarchy in parallel. Some branches intentionally will be developed earlier than other branches. For example, user or other external interfaces might be developed to permit early training

or hardware/software integration. Also, in many applications, requirements will become firm in certain areas before others. The areas covered by known requirements can usually become operational while the requirements are being developed for the others. Some segments, intended to support long-range requirements, may remain after the program is fully operational to serve as guides.

B.3 PROGRAMMING SUPPORT LIBRARY

A Programming Support Library (PSL) serves as a repository for data necessary for the orderly development of computer programs using structured programming technology. The data exists in two forms:

- o Data stored in machine readable form accessible by the computer
- o Corresponding data stored in hardcopy (human readable) form in project notebooks.

Included with a PSL are the necessary computer and office procedures for manipulating this data.

The purpose of a PSL is to support the program development process. This involves the support of the actual programming process and the management of the process.

Support of the programming process involves support of the design, coding, testing, documentation, and maintenance of computer programs and the associated data base definitions. A PSL provides this support through:

- o Storage and maintenance of programming data
- o Output of programming data and related control data
- o Support of the compilation and testing of programs
- o Support of the generation of program documentation.

A PSL must also provide some means of generating and maintaining itself.

Support of the programming management process also involves the storage and output of programming data. In addition, it involves:

- o Collection and reporting of management data related to program development
- o Control over the integrity and security of the data stored in the PSL
- o Separation of the clerical activity related to the programming process.

A PSL supports an approach in which people work on a common, visible product rather than on independent pieces. The programmers communicate through this product in carrying out programming and clerical interface activities. A PSL permits a programmer to exercise a wider span of detailed control and reduces explicit communication requirements. This makes it easier to bring new personnel onboard and to shift programmers from one part of the project to another. It also minimizes the preparation effort for technical audits. A secretary/librarian is responsible for maintaining the notebooks and archives of the PSL, and the programmers are responsible for their contents. This structure of responsibility permits standardization in project recordkeeping and ensures that the hardcopy listings in the library correspond to the most current version of the system.

A PSL system has four components:

a. Internal libraries

b. External libraries

c. Computer procedures

d. Office procedures.

The components of the system are interlocked to establish an exact correspondence between the internal (computer readable) and the external (programmer readable) versions of the developing system. This continuous correspondence is the characteristic of a PSL that guarantees ongoing visibility of the developing system.

Different implementations of a PSL exist for various computer and operating system environments used in system development. The fundamental correspondence between the internal and external libraries in each environment is established by the PSL office and computer procedures. The office procedures are specified at a detailed level so that the format of the external libraries will be standard across programming projects, and the maintenance of both internal and external libraries can be accomplished as a clerical function. The PSL computer procedures for each are expressly designed for easy invocation by secretary/librarian personnel so that their use is nearly fail-safe. A PSL is further discussed in Volumes V and VI of the Structured Programming Series.

B.4 PROGRAM DESIGN LANGUAGE

Top down structured programming concepts are now being extended to include the design of programs to be developed. Traditionally, narrative descriptions, decision tables, and flowcharts have been used in describing the design of a software system. These techniques are in the process of being supplanted by program design languages which are intended as a:

- o Vehicle to translate functional specifications into program design
- o Replacement for design logic flowcharts
- o Means for communication between technical and nontechnical personnel, designers, and developers.

An additional benefit is that a PDL has a more natural relationship to programming languages than traditional methods thereby facilitating the mapping of function into code.

PDLs, as currently practiced, are English-like and usually follow some semantical and syntactical conventions. The concepts of structured programming are applied in the form of basic control structures for logic flow and indentation. Top down programming is implemented by specifying in PDL the top level portion of the program and evolving the PDL into succeeding levels of detail. Considerable choices are left to the programmer in the selection of predicate and function descriptions which may be English statements in the computer language to be used for implementation or some combination.

The advantage of flowcharts and decision tables over straight prose is that the flowcharts and tables take advantage of two dimensions to show interrelationships which are obscure in the linear medium of prose.

A PDL is two-dimensional prose (three, if one counts segmentation) without the constraints of fixed-size boxes or table-entries. It can thus show the same interrelationships as the flowcharts in a clearer manner.

If used in the design phase, a PDL provides technical communication between designer and programmer. If used during implementation, verification for completeness and correctness is enhanced. At any level of the evolving program, design review and verification can be completed prior to commitment to source code. PDL is further discussed in Volume VIII of the Structured Programming Series.

B.5 DEFINITIONS

For this report, the following definitions will apply:

Program Design Language (PDL) -- A textual, English-like language describing the control structure and general organization of a computer program. The purpose of this tool is to facilitate the translation of functional specifications into computer instructions.

Programming Support Library (PSL) -- A repository for data necessary for the orderly development of computer programs using structured programming technology. The data repository is in two forms: data stored in machine readable form accessible by the computer and the identical data stored in hard copy form in project notebooks. A PSL also includes the necessary computer and office procedures for manipulating this data.

Structured program -- A program constructed of a basic set of control logic figures which provide at least the following: sequence of two operations, conditional branch to one of two operations and return repetition of an operation. A structured program has only one entry and one exit point. In addition, a path will exist from the entry to each node and from each node to the exit.

Structured Programming (SP) -- The process of developing structured programs. Associated with structured programming are certain practices such as indentations of source code to represent logic levels, the use of intelligent data names, and descriptive commentary.

Structured Programming Technology (SPT) -- A term which collectively references the following list:

- o Program Design Language (PDL) concepts
- o Programming Support Library (PSL)
- o Top Down Structured Programming (TDSP)

Structured segment -- A logically complete set of executable instructions constructed of nested structured programming figures. In addition to or in place of executable instructions, a structured segment may include nonexecutable instructions such as data declarations and descriptive commentary.

Structured source code listing -- A listing comprised of the following sections:

- o Section 1 contains the first executable structured segment (commonly referred to as the top level segment) as coded in the source programming language.
- o Section 2 contains all remaining structured segments. The structured segments are alphabetized by name. As in Section 1, each structured segment is represented as coded in the source programming language.
- o Section 3 contains the executing sequence among the structured segments.

Top Down Programming (TDP) -- The concept of performing in hierarchical sequence a detailed design, code, integration and test as concurrent operations.

Top down structured program -- A structured program with the additional characteristics of the source code being logically, but not necessarily physically, segmented in a hierarchical manner and only dependent on code already written. Control of execution between segments is restricted to transfers between adjacent hierarchical segments.

Top Down Structured programming (TDSP) -- The process of developing top down structured programs. Associated with top down structured programming are certain practices such as indentations of source code to represent logic levels, the use of intelligent data names and descriptive commentary. Top down structured programming requires top down programming as the primary implementation methodology.

REFERENCES

1. Gruenberger, F., "Program Testing: The Historical Perspective," Computer Program Test Methods Symposium, June 21-23, 1972, Program Test Methods, Prentice-Hall Inc., pp 7-10.

2. Hetzel, W. C., "A Definitional Framework," Computer Program Test Methods Symposium, June 21-23, 1972, Program Test Methods, Prentice-Hall Inc., pp 7-10.

3. King, J., "A Verifying Compiler," Courant Computer Science Symposium 1, June 29-July 1, 1970, Debugging Techniques in Large Systems, Prentice-Hall Inc., pp 17-40.

4. Good, D. I., "Provable Programs and Processors," National Computer Conference, 1974, pp 357-363.

5. Good, D. I., London, R. L., Blodsoe, W. W., "An Interactive Program Verification System," October 1974, pp 1-26.

6. Boyer, R. S., Elspas, B., Levitt, K. N., "SELECT -- A System for Testing and Debugging Programs by Symbolic Execution," Computer Science Group, Stanford Research Institute, Menlo Park, California 94025, October 1974, pp 1-46.

7. Robinson, L., Holt, R. C., "Formal Specifications for Solutions to Synchronization Problem," Computer Science Group, Stanford Research Institute, Menlo Park, California 94025.

8. Reifer, D. J., "Computer Program Verification/Validation/Certification," Technology Division of the Aerospace Corporation, TOR-0074 (4112)-5, May 1974, pp 1-36.

9. Keirstead, R. E., Parker, D. B., "On the Feasibility of Formal Certification," Computer Program Test Methods Symposium, June 21-23, 1972, Program Test Methods, Prentice-Hall Inc., pp 291-301.

10. LaPodula, L. J., "Reliability Modeling and Measurement," MITRE Corp., TR 2468, June 1973, pp 1-82.

11. Mills, H. D., "The Complexity of Programs," Computer Program Test Methods Symposium, June 21-23, 1972, Program Test Methods, Prentice-Hall Inc., pp 225-238.

12. Mills, H. D., "On the Development of Large Reliable Programs," IEEE Symposium on Computer Software Reliability, April 1973, pp 155-159.

13. Schneiderwind, N. F., "An Approach to Software Reliability Prediction and Quality Control," Naval Postgraduate School, Monterey, California, Fall Joint Computer Conference, 1972, pp 837-847.

14. TRW, "Software Reliability Study," Interim Technical Report, 74-2260.1.9-29, June 1974.

15. Sullivan, J. E., "Measuring the Complexity of Computer Software," MITRE Corp. TR 2648, June 1973, pp 1-41.

16. Brown, J. R., DeSalvio, A. J., Heine, D. E., Purday, J. G., "Automated Software Quality Assurance," Computer Program Test Methods Symposium, June 21-23, 1972, Program Test Methods, Prentice-Hall Inc., pp 181-203.

17. Keezer, E. I., "Practical Experiences in Establishing Software Quality Assurance," IEEE Symposium on Computer Software Reliability, April 1973, pp 132-135.

18. Hetzel, W. C., "Principles of Computer Program Testing," Computer Program Test Methods Symposium, June 21-23, 1972, Program Test Methods, Prentice-Hall Inc., pp 7-10.

19. Merten, A., Teichroew, D., "The Impact of Problem Statement Languages on Evaluating and Improving Software Performance," The University of Michigan, Ann Arbor, Michigan, Proceedings from the Fall Joint Computer Conference, 1972, pp 849-857.

20. Baird, G. N., "The DoD COBOL Compiler Validation System," Proceedings from the Fall Joint Computer Conference, 1972, pp 819-827.

21. Brown, J. R., Hoffman, R. H., "A Survey of Techniques and Automated Tools," TRW Systems, May 1972, pp 1-20.

22. Goldstein, A., Goldstein, B. C., Sapiro, A. J., "TMGP - Test Matrix Generating Program," TR 00.2353, IBM, August 1972, pp 1-20.

23. Hanford, K. V., "Automatic Generation of Test Cases," IBM Systems Journal, No. 4, 1970, pp 242-256.

24. Kosy, D. W., "Air Force Command and Control Information Processing in the 1980's: Trend in Software Technology," Rand Corporation, R-1012-PR, Santa Monica, California, June 1974, pp 1-177.

25. Modern Data, "Survey of Program Packages - Programming Aids," March 1970, pp 62-72.

26. Baker, F. T., "System Quality Through Structured Programming," AFIPS Conference Proceedings, Volume 41, Part 1, 1972 Fall Joint Computer Conference, pp 339-343.

27. Graham, R. B., Waunid, C. K., "A Survey of Systems and Techniques, Testing Validation of Complex, Interactive Applications," IRAD Task 1 Report, IBM Houston, 1974, pp 1-44.

28. Buckley, F. J., "Software Testing - A Report from the Field," IEEE Symposium on Computer Software Reliability, April 1973, pp 102-106.

29. Fagan, M. E., "Design and Code Inspections and Process Control in the Development of Programs," TR 21.572, IBM System Development Division, December 1974, pp 1-35.

30. Corrigan, A. E., "Results of an Experiment in the Application of Software Quality Principles," MTR-2874 (Volume III), June 30, 1974, pp 1-91.

31. Scherr, A. L., "Developing and Testing a Large Programming System - OS/360 Time Sharing Option," Computer Program Test Methods Symposium, June 21-23, 1972, Program Test Methods, Prentice-Hall Inc., pp 165-180.

32. Stucki, L. G., "A Prototype Automatic Program Testing Tool," McDonald Douglas Astronautics Company, Huntington Beach, California, Fall Joint Computer Conference, 1972, pp 829-836.

33. Stucki, L. G., "Automated Tools and Techniques Assisting in Software Development," McDonald Douglas Astronautics Company, 1973, pp 1-25.

34. Stucki, L. G., "Automatic Generation of Self-Metric Software," IEEE Symposium on Computer Software Reliability, April 1973, pp 94-100.

35. Stucki, L. G., Foshie, G. L., "New Assertion Concepts for Self-Metric Software Validation," McDonald Douglas Astronautics Company - West, WD2505, March 1975, pp 1-13.

36. Nelson, E. C., "A Statistical Basis for Software Reliability Assessment," TRW-SS-73-03, March 1973, pp 1-9.

37. Krause, K. W., Smith, R. W., TRW Systems, Houston, Texas, and Goodwin, M. A., Johnson Space Center, Houston, Texas, "Optimal Software Test Planning through Automated Network Analysis," pp 18-32.

38. Mills, H. D., "On the Statistical Validation of Computer Programs," FSC 72-6015, IBM pp 1-11.

39. Yelowitz, L., "Progress Report - Project No. 306 (Program Validation)," July 1973, pp 1-51.

40. Elmendorf, W. R., "Disciplined Software Testing," Courant Computer Science Symposium 1, June 29-July 1, 1970, Debugging Techniques in Large Systems, Prentice-Hall Inc., pp 137-139.

41. Freeman, P., "Functional Programming Testing and Machines Aids," Computer Program Test Methods Symposium, June 21-23, 1972, Program Test Methods, Prentice-Hall Inc., pp 41-47.

42. Drummond, M. E. Jr., "Simulation Techniques," Evaluation and Measurement Techniques for Digital Computer Systems, Prentice-Hall Inc., 1973, pp 145-201.

43. Stanley, W. I., Hertel, H. F., "Statistics Gathering and Simulation for the Apollo Real-Time Operating System," IBM Systems Journal, Volume 7, November 2, 1968, pp 8-15.

44. Department of the Army, "Management Information Systems - Policies, Objectives, Procedures, and Responsibilities," Army Regulation AR 18-1, pp 2-2 through 2-16.

45. IBM "Programming Project Management Guide," IBM Data Processing Techniques Manual - GA36-0005-1, 1974.

46. Head, R. O., "Automated System Analysis," Datamation, August 15, 1971, pp 22-24.

47. Martin, D. B., "Coding Hints for Large Systems," IBM TR00.1968, December 1969.

48. Green, J. H., "Coding Techniques for Virtual Memory, IBM, TR00.2232, June 1972.

49. Dijkstra, E. W., "The Structure of The Multiprogramming System," Communication of the ACM, Volume II, No. 5, May 1968, pp 84-88.

50. Mills, H. D., "Top Down Programming in Large Systems," Courant Computer Science Symposium 1, June 29-July 1, 1970, Debugging Techniques in Large Systems, Prentice-Hall Inc., pp 41-55.

51. IBM, "Structured Walk-throughs: A Project Management Tool," Data Processing Division, August 1973, pp 7-10.

52. IBM, "FSD Basic Programmer Training Guide."

53. Jelinski, Z., Moranda, P. B., "Application of a Probability - Based Model to a Code Reading Experiment," IEEE Symposium on Computer Software Reliability, April 1973, pp 78-81.

54. Graham, R. M., Clancy, G. J. Sr., DeVaney, D. B., "A Software Design and Evaluation Systems," Communication of the ACM, Volume 16, Number 2, February 1973, pp 110-116.

55. Bohm, C., Jacopini, G., "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," Communications of the Association for Computing Machinery, Volume 9, No. [illegible] y 1966.

56. Mills, H. D., "Mathematical Foundations for Structured Programming," IBM Corp., FSD 72-6012, February 1972.

BIBLIOGRAPHY

Bell, T. E., "Objectives and Problems in Simulating Computers," AFIPS Conference Proceedings, Volume 41, Part 1, 1972 Fall Joint Computer Conference, pp 287-297.

enson, J. P., "Structured Programming Techniques," IEEE Symposium on Computer Software Reliability, April 1973, pp 143-147.

Boehm, B. W., "Software Design and Structuring," TRW Systems, 197_, pp 1-24.

Boehm, B. W., "Software Reliability and Software Error Models," TRW Systems, 1973, pp 1-10.

Davis, R. M., "Quality Software Can Change the Computer Industry," Computer Program Test Methods Symposium, June 21-23, 1972, Program Test Methods, Prentice-Hall Inc., pp 303-311.

General Research Corporation, "RXVP: An Automated Verification System for FORTRAN," February 1975, pp 1-16.

Gruenberger, F., "Program Testing and Validation," Datamation, July 1968, pp 39-47.

Holland, J. G., "Acceptance Testing for Application Programs," Computer Program Test Methods Symposium, June 21-23, 1972, Program Test Methods, Prentice-Hall Inc., pp 263-274.

Howard, J. H., Alexander, W. P., "Analyzing Sequences of Operations Performed by Programs," Computer Program Test Methods Symposium, June 21-23, 1972, Program Test Methods, Prentice-Hall Inc., pp 239-254.

IBM, "Space Lab - Software Development and Integration Concepts Study Report," October 1973, pp 2-4 through 2-11.

Itoh, D., Izutani, T., "FADEBUG-I, A New Tool for Program Debugging," IEEE Symposium on Computer Software Reliability, April 1973, pp 38-43.

Kosy, D. W., "Approaches To Improved Program Validation Through Programming Language Design," AD-751 826, Rand Corporation, Santa Monica, California, July 1972, pp 1-25.

Kulsrud, H. E., "Extending the Interactive Debugging System - HELPER," Courant Computer Science Symposium 1, June 29-July 1, 1970, Debugging Techniques in Large Systems, Prentice-Hall Inc., pp 77-91.

Lindfors, R. S., "Design Validation Overview, Systems Programming and Analysis," IBM FSD Houston, 1973, pp 1-69.

Mills, H. D., "How to Write Correct Programs and Know It," IBM, FSC 73-5008, 1973, pp 1-26.

Mills, H. D., "Reading Programs as a Managerial Activity," working paper, March 1972, pp 1-10.

Myers, G. J., "Composite Design: The Design of Modular Programs," IBM TR00.2406, January 1973.

Naughton, J. J., "An Application Development," working paper, January 1973, pp 1-13.

Ng, E. W., "Mathematical Software Testing Activities," Computer Program Test Methods Symposium, June 21-23, 1972, Program Test Methods, Prentice-Hall Inc., pp 135-141.

Paige, M. R., Balkovich, E. E., "On Testing Programs," General Research Corporation, Santa Barbara, California, IEEE Symposium on Computer Software Reliability, April 1973, pp 23-27.

Prokop, J. S., "On Proving the Correctness of Computer Programs," Computer Program Test Methods Symposium, June 21-23, 1972, Program Test Methods, Prentice-Hall Inc., pp 29-37.

Ramamoorthy, C. V., Meeker, R. E. Jr., Turner, J., "Design and Construction of an Automated Software Evaluation System," IEEE Symposium on Computer Software Reliability, April 1973, pp 28-37.

Reifer, D. J., "Interim Report on the Aids Inventory Project," Technology Division of the Aerospace Corporation, SAMSO-TR-75-8, 1975, pp 1-68.

Rubey, R. J., "New Approaches for Software Validation," Naecon 72 Record, 1972, pp 252-257.

Supnik, R. M., "Debugging Under Simulation," Courant Computer Science Symposium 1, June 29-July 1, 1970, Debugging Techniques in Large Systems, Prentice-Hall Inc., pp 117-136.

VanNoot, T. J., "System Testing - Taboo Subject," Datamation, December 15, 1971, pp 61-64.

Vyssotsky, V. A., "Common Sense in Designing Testable Software," Computer Program Test Methods Symposium, June 21-23, 1972 Program Test Methods, Prentice-Hall Inc., pp 41-47.

Writtenbrook, W. K., "Testing a PL/I Structured Program," IBM, TR 54.041, December 1973, pp 1-10.

Wulf, W. A., "ALPHARD: Toward a Language to Support Structured Programs," Carnegie-Mellon University, Pittsburgh, Penna, April 1974, pp 1-17.

Yelowitz, L., "A Symmetric Top Down Structured Approach to Computer Program/Project Development," IBM, FSC 73-5001, IBM, 1973.

MISSION of Rome Air Development Center

RADC is the principal AFSC organization charged with planning and executing the USAF exploratory and advanced development programs for information sciences, intelligence, command, control and communications technology, products and services oriented to the needs of the USAF. Primary RADC mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, and electronic reliability, maintainability and compatibility. RADC has mission responsibility as assigned by AFSC for demonstration and acquisition of selected subsystems and systems in the intelligence, mapping, charting, command, control and communications areas.