MARCH 1976

NASA CR-144769

FINAL REPORT
A METHODOLOGY FOR PRODUCING RELIABLE SOFTWARE

VOLUME I

MCDONNELL DOUGLAS ASTRONAUTICS COMPANY

MCDONNELL DOUGLAS CORPORATION

MCDONNELL DOUGLAS

FINAL REPORT
A METHODOLOGY FOR PRODUCING RELIABLE SOFTWARE

VOLUME I

MARCH 1976

MDC G6210

PRINCIPAL INVESTIGATOR: Leon G. Stucki

LEON G. STUCKI
PROJECT MANAGER
AUTOMATED VERIFICATION SYSTEMS

OTHER CONTRIBUTORS:

PAUL MORANDA
GARY FOSHEE
MARJORIE KIRCHOFF
ROGER OMRE

APPROVED BY: Zygmunt Jelinski

ZYGMUNT JELINSKI
MANAGER
COMPUTER SCIENCES

MCDONNELL DOUGLAS ASTRONAUTICS COMPANY-WEST

5301 Bolsa Avenue, Huntington Beach, CA 92647

PREFACE

This report was prepared for Mr. Evmenios Damon, Mission Operations Computing Division, NASA Goddard Space Flight Center under Contract Number NAS 5-20781.

Requests for further information or assistance will be welcomed by McDonnell Douglas representatives:

- L. G. Stucki, Study Manager
  Huntington Beach, California
  Telephone: 714-896-3774

- M. W. Hogan, Contract Administrator
  Huntington Beach, California
  Telephone: 714-896-4856

CONTENTS

Section 1

INTRODUCTION

1.1 STUDY OBJECTIVE

The problem of producing reliable software has been with us for a number of years and yet until recently only portions of the problem have been attacked. For example, automated tools for analyzing program structures have been developed and loudly acclaimed by some while others promote tools for automated test case generation. The use of a theoretical approach to the problem has also been investigated but at no time has a comprehensive view of the total problem been taken. The purpose of this study is to perform for NASA an investigation into the areas having an impact on producing reliable software including automated verification tools, software modeling, testing techniques, structured programming and management techniques. This final report contains the results of this investigation, analysis of each technique, and the definition of a methodology for producing reliable software.

Task I - Automated Verification Tools (Appendix A)

- Investigation was made into the existence, availability and applicability of automated verification tools such as PET, ATDG, FORTUNE, etc.
- A comparative analysis was made and relative merits evaluated.
- Recommendations regarding development of new tools and modifications of the existing tools for NASA applications are included.

Task II - Software Modeling (Appendix B)

- Evaluation of the existing approaches to software modeling was made. The practical application of statistical techniques to software quality measurement was assessed.
- Correlation between programming structures and figure-of-merit indexes is considered among the major recommendations for future research.

Task III - Program Testing (Appendix C)

- Testing techniques currently available were evaluated.
- Problems of test case design were considered.
- Techniques for test design and optimization in relation to cost and software reliability are recommended for future research.

Task IV - Structured Programming (Appendix D)

- State-of-the-art structured programming applications were studied with specific consideration of higher level languages suitability to structured programming.

Task V - Program Management Techniques (Appendix D)

- Chief Programmer Techniques and Computer Program Management Techniques were studied and comparative analyses are made.

Task VI - Proving Programs Correct (Appendix E)

- A review of literature on the techniques of Proving Program Correctness was made. A glossary of terms with relevance to Software Reliability was developed. Techniques which may be applicable to Test Case Selection will be identified.

Final Report - Overview

This final report contains: an evaluation and assessment of the practicability of each available technique, a description of a methodology for producing reliable software, suggestions for automating portions of the methodology, and a comprenensive bibliography.

1.2 STUDY RECOMMENDATIONS

In surveying the various tools and techniques currently being used or developed for producing more reliable software, one quickly realizes the need for an overall methodology. A great deal of work has been put into various tools promising increased software reliability. At the same time a few well known individuals have been espousing the belief that by applying certain management and organization concepts programmers will produce error free programs. Despite these sometimes inflated claims, both the tool builders and the organization advocates have made positive contributions towards achieving more reliable software systems. However, it is now quite clear that an integrated methodology including sound organizational concepts and procedures with complementary support from automated tools offers an improvement.

With this philosophy in mind, this final report attempts to present an overall methodology for producing reliable software. Tools are incorporated into this methodology at numerous points. A series of appendices contain a significant amount of data on currently available tools and techniques.

The test tool survey contained in Appendix A is a "first". It is the first time that an attempt has been made at comparing a number of currently available tools and providing the reader with a means of examining the capabilities of all of the tools. A number of obvious omissions can be observed due to the reluctance of some tool builders to include their tools in such a survey. One very definite recommendation of this study is to expand this survey to include additional tools. Another area suggesting more research is the development of more meaningful measures for comparing the cost and performance associated with the use various tools.

With respect to the types of tools currently available, it is strongly recommended that static analyzer tools and dynamic execution analyzer tools be incorporated into a facility's general support software. These tools should be made available for general use and in connection with the suggested methodology contained in this report.

Two distinct areas of research mentioned in this report appear very promising with regard to tools. Appendix C describes the use of an embedded assertion language for bridging the gap between requirements specification and the functional testing of those specifications. This approach promises to make a major impact on the actual programming process. Appendix C also contains an interesting look into a promising means for rigidly examining various properties of a program and thereafter being able to informally prove the corresponding correctness of the selected properties. Additional research in these areas can lead to more powerful tools in the future.

Software modeling is another research area demanding additional effort. In particular, the refinement of error data collection process, the association of the error data with the original source of the anomaly, and the correlation of the degree of testing applied to a module of software and the number of errors associated with that module constitute ongoing research topics.

1.3 METHODOLOGY OVERVIEW

In most systems being developed today, the quality of the software is often the limiting factor of the total system quality. Rigorous quality assurance disciplines have been imposed on hardware for many years resulting in highly reliable hardware. These same disciplines imposed on the software have not achieved reliability of the software because, with the state-of-the-art tools, software cannot be tested as rigorously as hardware. Most often the end product is still less than satisfactory.

The problem is often the result of several factors:

- quality assuring activities are imposed late in the development cycle
- quality assuring activities are treated as unrelated activities
- quality assuring activities are not always the first concern of the software builder.

In the normal course of software development, the initial requirements analysis and development specifications are both incomplete and conflicting. If the inadequacies are not discovered and corrected, they are incorporated into the design. The poor design is then implemented in code. The problems are not detected until the testing phase which reveals the need for changes to the detailed design which in turn may cause changes in the requirements. This recognition of the problem late in the development cycle results in software redesign that is often difficult to incorporate into existing work, causing costly overruns.

Ideally, this iterative interaction of error detection and correction should be confined to successive phases. When this is true, the goal of ensuring the completeness of the requirements is pursued during the design phase. The requirements then should be completely and accurately defined and under formal controls by the time testing begins.

The achievement of quality software can be promoted by the application of a methodology that imposes quality producing activities on the development cycle.

This methodology consists of three interdependent sets of techniques:

1) software production techniques
2) software verification/validation techniques
3) software management techniques

Software production techniques include such items as:

- top down development
- structured programming
- use of a program design language (PDL)
- use of tools such as compiler writers, meta-assemblers and language preprocessors where applicable.

Software verification/validation techniques include such direct activities as:

- requirements analysis and feedback
- an assertion methodology for placing specification checks within the code
- code verification using walk throughs, standards checkers, execution analyzers, flowpath analyzers and debugging aids.
- program validation by module, acceptance and system integration testing.
- program certification

Software management techniques are the means by which the development is ordered and controlled, and by which visibility is provided into the status of progress and quality of the software development. These techniques include:

- configuration management
- program library control
- use of stringent documentation standards

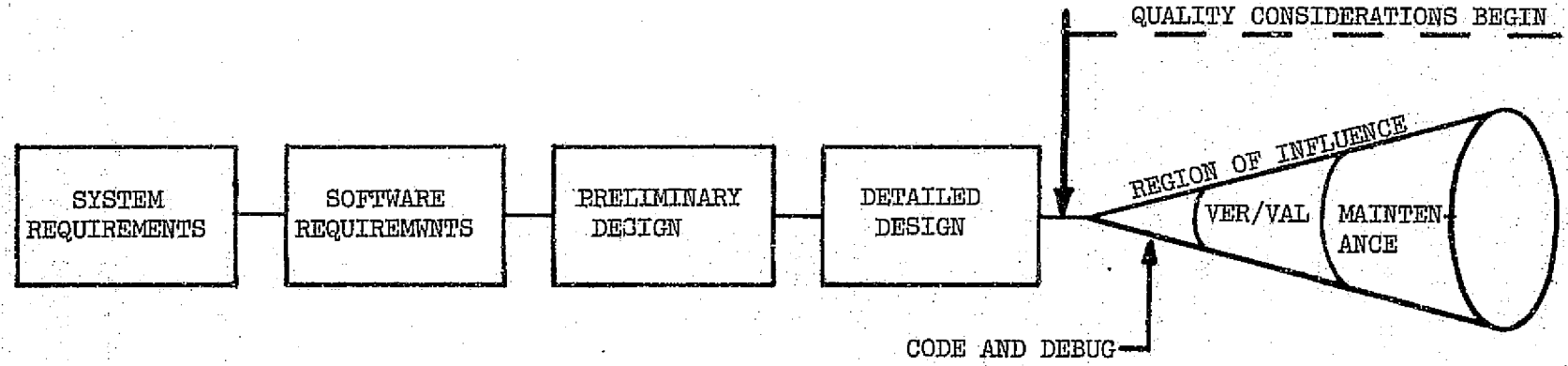
However, the presence of these activities does not in itself assure that quality will be achieved. The decisions on when to begin an activity and on the degree of the discipline with which it will be performed are crucial. Decisions that shape the project have an increasing influence on the development cycle as it progresses. When quality considerations are delayed in the development cycle as shown in Figure 1-1, their influence is limited. By the time coding and testing have started and project members are concerned with product quality, their efforts to achieve that quality are often restricted by past decisions.

Figure 1-2 shows the effect of including quality considerations in the criteria for the software decisions in all phases of development, so that each phase contributes to the quality goal. Since the development of a software system is basically a human activity, it is imperative that information about the deveopment is constantly and clearly fed back to the developer and the user right from the beginning.

The immediate imposition of standards to assure quality documentation provides confidence that the intent of the documents is accurately communicated to those who must rely on those documents.

Early review and analysis of specifications allows early decisions on redesign so that changes are preventive rather than curative. The test and evaluation function begins with specification analysis and constantly impacts all phases. Early considerations of requirements testability reduces likelihood of costly impacts in the later phases of development.

The early determination of requirements for production and verification tools allows time for them to be procured (built, purchased or leased) and checked out before they are needed. Reliability measurement definitions at an early stage dictate the data to be collected and applied during development.

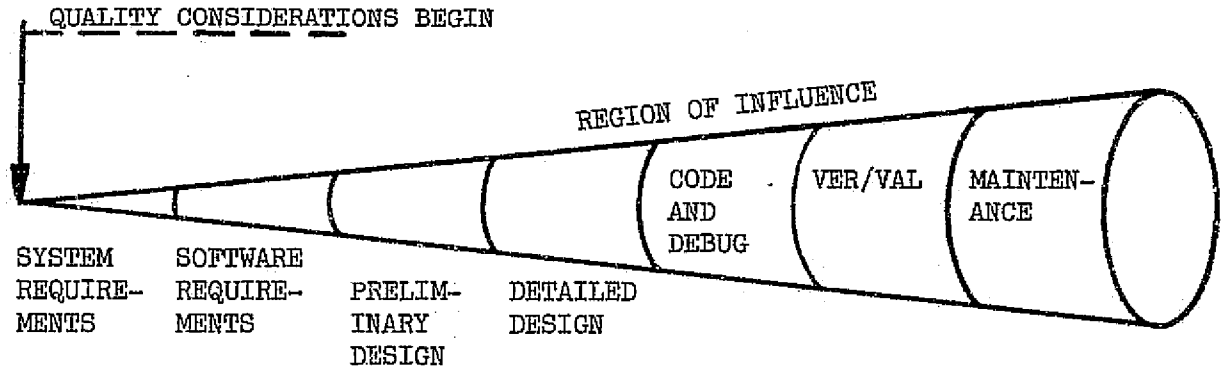


SOFTWARE PRODUCTION TECHNIQUES
- BOTTOM-UP DEVELOPMENT
- NORMAL PROGRAMMING METHODS
- DEVELOPMENT TOOLS

SOFTWARE VERIFICATION/ VALIDATION TECHNIQUES
- REQUIREMENTS ANALYSIS AND FEEDBACK
- AUTOMATED VERIFICATION TOOLS
- PROGRAM VERIFICATION/VALIDATION

SOFTWARE MANAGEMENT TECHNIQUES
- CONFIGURATION MANAGEMENT
- PROGRAM LIBRARY CONTROL
- DOCUMENTATION STANDARDS CONTROL

Figure 1-1
EFFECT OF DELAYING QUALITY CONSIDERATIONS



SOFTWARE PRODUCTION TECHNIQUES
- TOP-DOWN DEVELOPMENT
- STRUCTURED PROGRAMMING
- DEVELOPMENT TOOLS

SOFTWARE VERIFICATION/ VALIDATION TECHNIQUES
- REQUIREMENTS ANALYSIS AND FEEDBACK
- AUTOMATED VERIFICATION TOOLS
- PROGRAM VERIFICATION/VALIDATION

SOFTWARE MANAGEMENT TECHNIQUES
- CONFIGURATION MANAGEMENT
- PROGRAM LIBRARY CONTROL
- DOCUMENTATION STANDARDS CONTROL

Figure 1-2
EFFECT OF EARLY QUALITY CONSIDERATIONS

Configuration management and program library control processes instituted early reduce confusion in the detailed design and coding phases and allow development to progress in an orderly fashion.

The traditional approach to software quality assurance has been to have the quality functions such as testing, configuration management and program library control be performed by the software builder. There are several inherent disadvantages to this approach. The most obvious disadvantages are the development priority structure of the builder, which is different than the priorities of an independent evaluator, and the use of personnel to whom the quality assurance functions are of secondary interest.

Independent evaluation and monitoring provides an unbiased and effectual approach to software quality assurance. This assurance can be provided by imposing a quality producing methodology for controlling and validating the software at every stage of its development.

The functions required to achieve reliable software must be viewed as totally interrelated functions. The system cannot be used with confidence until it is well tested. It cannot be well tested without a comprehensive analysis of the requirments. The verification of requirements using test tools cannot be performed adequately without well checked out tools which must be identified in the requirements stage. The testing also is virtually meaningless without assurance of the integrity of the program library, which depends to a large extent on the effectiveness of the configuration management of the software. And finally, good documentation is required to provide feedback, visibility and assurance that these functions are effectively implemented.

The formulation of a methodology to produce reliable software for NASA Goddard considers the interrelationship of quality producing functions, the decisions that cause the imposition of these functions at the proper time, and the enforcement of the standards and controls that maintain the benefits of each function.

Section 2

TERMINOLOGY

Throughout the literature bearing on the subject of reliable software, there is considerable disagreement on the meaning of several key terms. To provide a consistent base for understanding this document, the following definitions have been selected for these terms. Although the list may appear to be elementary, it is intended to eliminate some of the ambiguity of meaning.

Software	-	the computer programs with their associated data bases, job control language and documentation.
Software quality	-	the set of properties of the software that characterize how well it works, how easy it is to use, and how easy it is to maintain.
Software reliability	-	(definition by Schick and Wolverton[1]) "the probability that the applications program, together with its supervisory program, data bases, and computing environment will perform its intended functions at the time when those functions are needed by the customer".
Verification	-	The process by which the set of specifications and/or axioms describing the nature of a problem and its environment are checked for completeness, consistency, and systematically compared with the resulting software representation. Verification addresses functional correctness and usually involves a great deal of manual effort. Automation of some of the comparative and analytical functions are currently being researched.
Validation	-	The process of inspecting software behavior in the operational environment (i.e., hardware, software, data sources, man-machine interface) and determining that the software will in fact perform its proper function. While verification attempts to build convincing arguments for the "correct" representation of a problem, validation addresses the questions:

1) Is the software really solving the right problem?

2) From an operational point of view is the software useable?

Debugging - the process of finding and correcting errors that are syntactic or structural in nature (not specifically associated with the verification and validation of basic program functions) and that prevent successful execution of the software.

Development testing - confirmation by the programmer that a completed software module performs as intended using a trivial test case. It is an informal test carried out in the software configuration currently being developed.

Unit testing - generally synonomous with development testing.

Module testing - confirmation that a completed module performs as intended when subjected to a comprehensive set of test cases in the software configuration currently being developed.

Integration testing - confirmation that a total hardware/software system performs as intended when the entire system is executed in a test environment.

Operational testing - the process of validating the system by exercising it for a given period of time in a user environment, using test procedures designed to exercise it comprehensively.

Acceptance testing - the testing performed with a set of test cases designed to verify that the completed system performs in accordance with specified acceptance criteria.

Baseline testing - confirmation that a completed system continues to meet the critical requirements during maintenance of the software. The test cases used are designed to exercise all software critical to the system performance.

Program Proving - The set of formal mathematical and informal quasi-mathematical techniques, often semi-automated, for checking the consistency of program specifications, axioms, and program code. Program proving is often equated with verification in academic circles. For purposes of this study, however, verification will be defined to include program proving techniques together with other less mathematically rigorous techniques for checking consistency and completeness.

Certification - the process of placing an authoritative stamp of approval on a software system. Certification implies that a software system has been subjected to an adequate level of testing and recommends its operational usage. Certification should include adequate verification and validation of the software although in current practice this is often not done.

Reference

1. G. J. Schick, R. W. Wolverton Assessment of Software Reliability.. Proceedings of German Operations Research Society, September 1972.

Section 3

SOFTWARE MANAGEMENT TECHNIQUES

3.1 GENERAL

The effective management of software development is of primary importance in producing reliable software. There are several techniques that will provide the controls that help guarantee orderly and efficient development.

Most of these techniques are used in some form at NASA Goddard. This section discusses enhancements of the techniques currently used that may offer greater visibility and more reliability.

The Programmer's Reference Manual provides standards and conventions for programming and documentation to ensure consistency of the product. It provides information to the programmer about the use of the system hardware and software that is installation-peculiar, and includes the detailed use of procedures that maintain control of configured libraries.

It also contains the procedures for designing effective development tests to be performed by the programmer.

Configuration management is discussed in relation to NASA's current procedures. A technique for maintaining automatic configuration status is offered using tools that will allow traceability of changes to the statement level. The capabilities of this type of tool include those already existing in the updating program used by NASA Goddard. The enhancements that provide the additional capabilities are described in detail in this section.

The establishment, updating, release, and maintenance of controlled libraries are affected by the techniques that affect configuration management. While the program library control and configuration management are discussed separately, they are interdependent.

The discussion of the software development organization stresses the use of two techniques: 1) independent analysis, review, testing and evaluation, and 2) the team approach to software development.

The last paragraphs discuss some techniques that help in organizing and measuring the development of the project. The selection of tools and techniques and the appropriate time to apply these tools and techniques are part of establishing the goals for project development. These tools and techniques are used both in the software production and software verification/validation areas.

3.2 SOFTWARE DOCUMENTATION

3.2.1 Programmer's Reference Manual

The programmer's reference manual contains the information a programmer needs to know about the environment in which he must work that is facility oriented. The purpose of this manual is to supplement information supplied in vendor manuals. The programmer's reference manual serves several functions.

1. It provides information about the resources available within the computer facility. These resources include the hardware configuration, the utility libraries, and the computer room and scheduling operations.

2. It contains the standards and conventions to be followed to insure standardization and completeness of the finished software projects. This includes programming standards and conventions, and documentation standards.

3. It prescribes the procedures to be followed in using the working and controlled libraries, and the procedures for installing new and modified code in a controlled environment.

4. It describes the recommended tools, and techniques to be used in checking out the code. This includes debugging aids, static and dynamic analysis tools, and development test techniques.

NASA Goddard has published a software standards guide to be used by in-house and contractor personnel using the IBM 360 facility. This document contains some of the information that should be present in a programmer's reference manual. This methodology recommends other information for inclusion.

1. Software Development Notebooks should be used and maintained by the programmer and the librarian. This technique is discussed in detail in Section 3.2.2.

2. Top-Down Development is a design technique to be used for organizing the development of a system. It is discussed in detail in Appendix D.

3. Tools and techniques are available that will aid FORTRAN code to be structured. These tools and techniques are discussed in detail in Appendix D.

4. The use of software production, testing, and documentation tools should be included. Candidate tools are discussed in Appendix A.

5. The programmer's reference manual should contain a section outlining the procedures to be followed to insure that a meaningful development test has been designed and performed. Development testing is discussed in detail in Section 4.3.1.

6. The development and maintenance of a well-controlled software system requires that the programmer understand how to effectively use the working and controlled libraries containing the system being developed or in operation. Program library control is discussed in Section 3.3.

7. This methodology recommends the team approach to program development. The responsibilities of each member of the team and his interfacing requirements should be included in the programmer's reference manual. The team approach to program development is discussed in Appendix D.

3.2.2 Software Development Notebooks

The use of Software Development Notebooks forces attention to every aspect of the development of software routines. The notebook provides a guide to and a record of specific programming activities and is used to assist in program documentation.

The notebook concept has been implemented at MDAC and TRW for the Site Defense Program (DSP) software development with a great deal of success. TRW calls the notebook "Unit Development Folder" (UDF).

R. D. Williams[1] describes the advantages of using their UDF as more than being a collection point for all pertinent development and test information. It ensures that documentation is updated as part of the development activity. It serves as a tool to help foresee impending difficulty in time to avoid it. It helps in obtaining meaningful estimates through direct involvement of project personnel in scheduling their own work, in accomplishing continuous monitoring and accurate reporting, in avoiding a proliferation of phantom problems described by Brooks[5], and in making effective use of time in updating plans or initiating corrective action at a time when it can do the most good.

3.2.2.1 Software Development Notebook Standards

A Software Development Notebook should comply with the following standards:

- Each module developed requires a notebook.
- Initially, each notebook will be assigned to one programmer.
- In late stages of program development, more than one programmer may have simultaneous responsibility for a notebook (module).
- A programmer may have simultaneous responsibility for more than one notebook.

- Changes to items 1-5 (see Figure 3-1) of a notebook at any time requires authorized approval.

- Changes to all other items in a software notebook after the routine has been placed in a controlled library requires authorized approval.

3.2.2.2 Software Development Notebook Contents

Figure 3-1 shows the cover sheet of a sample software development notebook.

	Due Date	Date Completed	Originator	Reviewer
1. Requirements Specification				
2. Design Description				
3. Functional Flow Chart				
4. Interface Description				
5. Assumptions and Constraints				
6. Module Code				
7. Development Test Case Descriptions				
8. Review of Development Test Cases				
9. Test Case Results				
10. Detailed Flow Chart				
11. Updated Design Description				
12. Program Library Control				
13. Discrepancy Report File				
14. Sign-off Completed Routine				

Figure 3-1. Development Notebook Cover Sheet

Table 3-1

DEVELOPMENT NOTEBOOK CONTENTS

1. Requirements Specification	This is the written material that describes the requirements on the routine; it tells what the routine shall do and how well it must do it.
2. Design Description	This is an English language description of how the routine shall perform. It is a description of the design that is being proposed to satisfy the requirements specification of 1 above.
3. Functional Flowchart	A flow diagram of the design described in 2 above.
4. Interface Definition	A list of all externally provided inputs and all generated output destined for use by other routines.
5. Assumptions and Constraints	A description of how the routine is invoked, how the called routines are used, the timing constraints, estimated core requirements and any unique conditions or assumptions associated with the routine.
6. Module Code	The initial routine code (a listing) which will be updated throughout the development period and which will keep pace with the code throughout its development.
7. Development Test Case Descriptions	A description of all development test cases which are to be used to checkout the routine and the results that can be expected from these test cases. Testing should be based upon both the functional capabilities list and the requirements specification of 1 above. Expected test case results will be included in the folder in advance of running th tests.

Table 3-1

DEVELOPMENT NOTEBOOK CONTENTS (Continued)

8. Code and Test Case Review	An engineering review (by someone other than the developer) to determine that the routine code will perform as defined in 1 above and that the proposed test cases do satisfactorily demonstrate this capability. This review is held in advance of actual testing.
9. Test Case Results	A compilation of all test case results to demonstrate that the routine is debugged and that routine development testing is complete. A listing of the debugged routine is to be included.
10. Detailed Flow Diagram	One which thoroughly details the delivered routine.
11. Updating Design Description	An updated/revised item 2 (corresponding to item 10 above).
12. Program Library Control	The point at which the completed code is entered into the controlled library. This is at the completion of the development testing, after the code and test case review.
13. Discrepancy Report File	Copies of every discrepancy report that required modification of the routine, or its documentation. They are added as they are resolved.
14. Sign-off Completed Routine	A formal acknowledgement that the routine is accepted for installation. For both new and modified systems, this is after formal qualification test, at the time the system is released for operation by Release Control.

3.2.2.3 Establishing, Maintaining and Using the Notebook

The procedure for preparing and using the notebook is as follows:

1. The routines are assigned with a budget allocation and final delivery date. The appropriate reviews are then allocated.

2. Each routine assigned to a programmer/analyst has its own notebook with a cover sheet. At the time of assignment the programmer negotiates his final date. Any schedule discrepancies are resolved and the resolved final date is entered opposite the item on the notebook cover sheet, "Updated Design Description" (see Figure 3-1).

3. The programmer is then required to plan his activity and schedule the various other items on the cover sheet. His supervisors now have incremental visibility into all the development steps of the software development activity.

4. The notebooks are always kept in one of two places, a file in the programming office area or the programmer's desk. If on the programmer's desk, then they will be signed out from the manager's file.

5. The notebooks will be available for review by authorized personnel at any time that they are not in use by the programmer.

6. The programmer has the responsibility to update the cover of the notebook to reflect the current status of its contents. The programming manager has the responsibility to review the contents and approve the cover sheet.

3.3 SOFTWARE CONTROL

Software control is achieved by the implementation of three interrelated disciplines.

1. Configuration management which is the day-to-day monitoring and control of the computer program configuration items (CPCI's) of which a system is composed. These items include the computer program and the associated documentation.

2. Release Control which is the process of closing out and turning over the software and related documentation. Software that is released consists of the computer program (source and object code) and supporting computer listings. The related documentation consists of the functional and detailed specifications and the User's Manuals. Each release establishes a new baseline for the product against which future modifications are made.

3. Program library control which is the process of establishing and maintaining a program library in which every statement is known, documented and traceable to the justification for its existence. It forms the basis for the configuration management of the code, the testability of the code, and the assurance that the integrity of released code is achieved.

The following paragraphs discuss these disciplines in reverse order.

3.3.1 Program Library Control

GSFC currently maintains three types of program libraries requiring control.

1. A permanent, distributable library containing all programs developed at GSFC.

2. On-line libraries containing source code, load modules, data bases, etc., of operational on-line systems.

3. A source and document library setup for developing large pieces of software.

The first two libraries contain only completed code already released. The third library is established and used during development.

This methodology discusses the recommended system for program library control that builds upon the system currently used at GSFC. It is supported by a tool that performs the functions that are performed by the utility in use at GSFC to automatically update and compile the programs, but in addition maintains the configuration status of the software and provides traceability of each statement back to the justification for its existence. One such tool has been developed for the HQ Space and Missile Organization (SAMSO) at Los Angeles and could be made available for use at GSFC.

3.3.1.1 Working Libraries and Controlled Libraries

During the development of a system, the necessity to continually work with existing baselined code requires the co-existence of working and controlled libraries. With the implementation of top-down development, this is true even with systems for which the greatest part is still in the form of dummy stubs.

The establishment of a controlled library consists of defining each routine, macro, data base segment, etc., as a configured item, assigning unique names and configuration identifiers, initializing the release and modification number of the tested code, and creating a permanent, protected library. Once the code is placed on the controlled library, it is established as baselined. Master and critical copies of the baselined code are created.

The use of the recommended update/compile program allows programmers to easily create working libraries for the development and debugging of new program segments or modifications to existing program segments.

Selected parts of the baselined source may be copied, updated and stored in a working library, either from TSO or from card decks. The listing provided by the update program should contain the configuration status information and identify the source of justification for the code (e.g., problem reports or design specs).

The programmer is forced to be aware of the version of the code with which he is working. He must input the current configuration revision symbol and is returned an error message if it is incorrect. This helps insure that he is not updating a different set of code than he thinks he is.

A parallel copy of the baselined library is maintained by a central group whose function is to maintain decks, memos, documentation, etc., and serve as the focal point for information about the program status. This library should be controlled to the extent that all changes and additions must be reviewed and approved before entry, documented both internally and externally, the configuration status maintained, and indepth testing performed against a known configuration.

The programmer's private working library is uncontrolled. The update program allows him to easily create a new file for new code or to copy into a working data set the code to be modified. He may then develop his code, debug the affected routines, link the new routines to the parallel system, perform development tests, and execute against a benchmark test without disturbing the basic system.

Control begins when the new approved code is placed into the parallel library.

At the point of making a formal modification and release, the current procedure is to freeze the update in the parallel library, test the frozen system, and when accepted, the new library is renamed and placed into operation.

If the changes are significant in number and/or complexity, it is recommended that the new system be completely rebuilt at the time of the update freeze. This means that the old system is copied to a new library, and all changes (including new and deleted routines) are made at one time to this library. This insures that every line of code in the new library is known, accounted for, and justified. It removes the possibility of a change of code being made that is undocumented or unjustified. The update program places all of the changes for a mod into a mod packet and writes them into a file on the end of the tape containing the updated source. This way, every change making up a modification is permanently stored in a readily retrievable form. In addition to the advantage of maintaining this record for historical purposes, it allows easy reconstruction of a change that must be backed out if an error is found after the formal mod is made.

3.3.1.2 Release Control

The release of a controlled system implies that it has been rigorously tested, completely documented, its configuration confirmed, all approvals have been obtained, and a master copy of the system (in both source and object form) placed in bond by a quality assurance organization.

The release function formalizes the installation of a new or modified system. It assures that the catalogued procedures are updated to reflect the new release, thereby minimizing the inadvertant use of a prior library when future changes are to be made or tests are to be performed.

The release control function is also concerned with protection of the operational system. Release control personnel should be the only personnel authorized to obtain a master copy of the system from Q.A. bond and restore the system online.

A Data Release Authorization is issued at the time of release and identifies the following:

1. Sequence number of release items
2. Mnemonic name of volume or document number
3. Current revision number
4. Security classification
5. Volume number, e.g., tape reel number
6. Program identification number
7. Next higher level of program
8. Responsible signatures

3.3.2 Configuration Management

Software configuration management is the day-to-day monitoring and control of the computer program configuration items (CPCI's) of which the system is composed. These items include the computer programs and the associated documentation.

The normal configuration management discipline is applied to the software. This discipline consists of four functions:

- configuration identification
- configuration change control
- configuration status accounting
- configuration audit

3.3.2.1 Configuration Identification

Configuration identification is a system of computer program identification numbers and document identification numbers that will identify all configurable items.

The configuration of a computer program should be documented in the specifications. The required configuration is identified in the design specification, and the achieved configuration is identified in the post development documentation.

Identification of the configurable software items for use with (or without) the update/compilation program consists of a comprehensive identification scheme. The following example shows one proposed method for accomplishing this goal:

System	-	A one to eight character name followed by a one character release number and one character modification number.
Programs	-	A filename of one to eight characters defining a file of one or more card decks. A deckname of one to eight characters defining a deck containing a subroutine, macro, data base, etc. A revision symbol of two characters beginning with AA. - Listings show the same configuration identification as the programs.
Documents	-	A one to eight character program document identifier followed by a two character revision symbol.

Documents to be assigned configuration identification numbers include:

- Part I CPCI (Functional) Specification
- Part II CPCI (Detailed Design) Specification
- Interface Specification
- Software Development Notebooks

3.3.2.2 Configuration Change Control

Configuration Change Control applies to all changes to configured software and documentation after the configured items are initially released.

All proposed changes to approved baselines are assessed, reviewed, and evaluated by the Change Control Board (CCB). The CCB is composed of representatives of the Design Group.

Actions by the CCB include verifying compliance with contractual requirements and assuring the identification, evaluation and consideration of the technical reasons for the change(s). The CCB guarantees that only those changes for which a requirement exists, or which offer a significant benefit to the program, are initiated. Members of the CCB determine the impact that proposed changes will have in the areas of cost, production, reliability, maintainability, producibility, logistical support and specifications.

Proposed changes are evaluated as they are proposed. The changes to be incorporated are recorded and collected together until the update freeze before the formal mod. At that time all incorporated changes are submitted as a unit to the CCB for final approval.

Proposed changes are initiated on a software problem report. The software modification report records the fix or improvement that was made. An engineering change proposal is used to formally submit the collected changes to the CCB for final approval. A specification change notice is used to record changes to specifications.

The CCB meets on a periodic basis or when a major change or improvement must be evaluated. These meetings are supplemented with separate individual meetings for review and action on individual problems that must be addressed.

Minutes of CCB meetings contain the transactions and assigned action items. They are not authorizations for changes, but are for historical and administrative purposes.

The problem reports being reviewed and any instruction for their disposition are attached to the minutes, signed by the CCB chairman, and distributed to all CCB members plus any others affected by the decisions.

All proposed changes must be addressed by the CCB. The decision to implement or not implement the change is made and recorded. The responsibility for investigating any unresolved changes is assigned by the CCB to a person who will obtain the information necessary for a decision to made.

Changes procedures apply to documentation as well as code.

3.3.2.3 Configuration Status Accounting

Configuration status accounting is the recording and reporting of the status of the system's configuration. The purpose is to know exactly what the current configuration is, and how it was achieved.

Configuration status accounting includes reporting and recording:

- the initial configuration identification
- the proposed changes to the configuration
- all approved changes to the configuration

As the initial configuration identification is updated, records are maintained that provide traceability of software problems or improvements from their inception through the corrective action to their incorporation into the existing system.

Status accounting applies to all controlled program documentation, the software development notebooks, and the code. Logs are maintained on the receipt, identification and disposition of all change forms. These include software problem reports, engineering change requests and specification change notices.

Status reports are published periodically. The information is retrieved automatically by report generators from the files maintained on a mass storage volume.

Other files contain the:

- history of all revisions and changes to each specification
- history and content of all problem reports

The status reports contain:

- history and status of specification changes
- summary of the status of all problem reports acted upon by the CCB, including disposition and schedule of change.
- status of any individual status report
- listing of the current computer program configuration

The above status reports can be obtained in a number of formats using various sorting criteria. This provides optimum visibility in any desired area.

3.3.2.4 Configuration Audit

Configuration audit consists of

1. a series of reviews of the requirements, the design, and the final baseline qualification, and

2. an audit of the functional and physical configuration of the system.

The purpose of the specification reviews is to confirm the presence of the information, clearly and accurately stated, necessary to develop the software that meets the requirements. Any problems detected are documented and presented for resolution.

The reviews serve to systematically evaluate the developing system and the end product in respect to its conformance with the requirements. Baselines are estabilished for the requirements, the design and the implemented code and documentation. The documentation is reviewed to assure that it accurately describes the product.

The audits assure that the configuration of the system is compliant with the requirements both functionally and physically.

3.4 SOFTWARE DEVELOPMENT ORGANIZATION

This methodology recommends two approaches to software development organization.

1. independent review, analysis and evaluation of the requirements and design, and independent test and evaluation of the completed product.

2. a team approach to software development.

The concept of the performance of verification/validation and control functions (which include the verification/validation of the requirements and the design) by personnel who did not specify, design, or build the system and who are specifically skilled in the areas of analysis and evaluation is becoming more widely accepted. There are two major advantages. One is the elimination of the logical bias inherent in having the designer/implementer perform these functions. The other is that the functions are performed by personnel to whom this discipline is of primary interest rather than secondary.

The concept of a team approach to software development has been widely discussed in the literature. This methodology offers a team approach that may by necessity draw the team members from various organizations.

3.4.1 Independent Requirements Analysis

The review and analysis of the requirements specified at both the functional level and the detailed design level by a group of one or more analysts who did not participate in the requirements definition or generation is recommended.

The review and analysis of specifications are critical functions. Initial specifications often contain ambiguities and are not always complete.

The independent analysts review the specification, talk to the requesting organization to determine if the specification accurately states the requirements, talk to the implementers (in-house or contractor personnel) to determine if their interpretation of the requirements is the same as the requestor's interpretation, and act as coordinators to resolve any inadequacies and conflicts in the requirements.

Prompt feedback must have a high priority to assure that design changes can be made at the earliest possible stage of development.

The independent analysts review and analyze the specifications for

- useability
- completeness
- clarity
- continuity
- uniformity
- traceability
- testability

The results of the analysis are presented to the approval authority, at preliminary design reviews and critical design reviews.

Requirements Analysis and Feedback is discussed in detail in Section 4.1.

3.4.2 Independent Test and Evaluation

The design and performance of test and evaluation functions by an independent organization is recommended.

For a new system, the test engineers review the requirements of the system and design asserted test criteria with accompanying test cases that will demonstrate that the implemented system performs as specified.

In the case of existing systems the independent test engineers review the improvements and corrections to be incorporated in the modifications, and design asserted test criteria with accompanying test cases that will demonstrate that the modifications cause the system to perform according to the requirements as well as demonstrate that the unmodified parts of the system are not degraded.

Asserted test criteria are documented in the test plan while also being placed within the respective programs using the embedded assertion language techniques described in Appendix C.

The test cases are executed on an informal basis until the formal mod is made, the discrepancies are recorded and given to the designers/programmers for correction, the test data is evaluated to assure that the test objectives are being met, and a formal test of the system is made. A final test report is prepared attesting to the extent to which the requirements are met as demonstrated by the test effort.

If certification is required, the test report is the basis for certification by the independent test and evaluation group.

Testing, evaluation and certification are discussed in Section 4.3 and 4.4.

3.4.3 Team Approach to Programming

In formulating a methodology for producing reliable software serious consideration should be given to the use of a team approach. Rather than recommending a specific approach, however, it is suggested that teams be tailored to the size of a project and the operational environment available at the developing location. A degree of flexibility should be provided in establishing the exact make up of a programming team. Past experience has shown that blind adherence to a reportedly "ideal" team organization can be counter productive. Various team approaches have been advertised widely in the literature. A discussion of several of these organizational schemes is contained in Appendix D.

3.5 PROJECT DEVELOPMENT PLAN

The development of a brand new system or the incorporation of a major function into an existing system needs to be carefully planned. The development plan should establish the goals of the project in terms of its purpose, and its function. It should define the methodology to be used for the implementations, and the controls to which the development is to be subjected.

3.5.1 Establishing Goals

In considering the development of a system with quality considerations built in, it is necessary to understand the scope and purpose of the total system at the very beginning. The top-down development of the system structure allows early visibility of the entire system, and a more accurate assessment of realistic performance and scheduling criteria at the outset.

The performance criteria are stated as requirement sets that satisfy the purpose of the system. The scheduling criteria are stated as milestone sets that satisfy delivery requirements. In general, the lack of early quality-producing considerations will adversely impact the schedule, since potential problems at the beginning become real problems later on, requiring correction time that might have been avoided.

Therefore, the goals to be established are those that will cumulatively result in a reliable system. Some of the major goals are:

1. produce a complete and consistent preliminary design that contains[1]:

    - major software functions which either directly correspond to or can be easily traced to explicit software requirements.
    - a set of test criteria which can be used in the formulation of a comprehensive test plan for validating and verifying the completed system.
    - a complete picture of the overall structure of the software system.
    - enough detail regarding the allocation of functional processing requirements to designated software elements to support a thorough and credible demonstration of design feasibility and validity.

2. produce a detailed design that:

    - is an extension of the top-down design concepts incorporated into the preliminary design.
    - expands the test criteria to be associated with the validation and verification of the resulting system.

- provides for a supervisory control routine for the implementation of each functional capability.

- provides clear traceability back through the preliminary design to the requirements and forward into the code and as-built documentation.

3. define and systematically carry out a series of reviews designed to:

   - create mutual understanding of the requirements by the requestor, the designer, the implementer, and the tester (design reviews).

   - communicate the existence, evaluation and correction of problems or potential problems (CCB).

   - communicate the configuration status of the developing system in relation to its specified configuration (CCB).

   - confirm that the code that implements the system will perform the functions required of it (walk through).

   - assess the extent to which the tested system meets the requirements specified for it (formal qualification review).

4. provide the designers, programmers, and testers with the aids (in the form of tools, guidelines and standards) that will help them to achieve and verify quality inthe product. Of particular importance is the timely availability of these aids.

5. Establish and maintain the controls that will preserve the integrity of the system at all stages of its development.

6. Motivate the personnel by:

   - providing them with opportunities to be flexible and innovative within the bounds of the controls placed upon the software development.

   - involving them directly in the scheduling of their own work as part of the Software Development Notebook concept.

There are a number of other goals that could be defined, but they can be classified into subsets of the above goals. For instance, the subject of documentation is not directly addressed, but is defined by the standards in item 3, and is procedurized in the controls in item 4.

In addition, the budget and milestone goals are not addressed, as their net effect on reliability generally comes about indirectly by causing pressure if estimates were not meaningful, or by avoiding pressures if they were.

3.5.2 Selection and Use of Aids

The selection of proper aids to be used during design, implementation and testing is critical to achieving and maintaining quality in large systems. Even more crucial is their availability at the time they are to be used. Availability of tools and techniques includes adequate procedures for their use. Tools that must be developed must be designed early enough to be built and thoroughly tested before they are used. Tools that are procured must be installed and checked out in the environment in which they are to be used.

The size and complexity of the system being developed, the hardware/software support resources available, and the tradeoff of value vs. cost are all factors to be considered in the selection.

Some tools and techniques can be advantageously applied to almost any system. For example, the walk-through technique is valuable even on small "one-time-only" programs, the difference being a matter of degree.

For the more trivial programs, an hour spent by another programmer informally reading the code may save several debugging runs, while on a large complex system public presentation of the code may be the best approach. Both are applications of the walk-through.

Obviously, some tools must be selected to appropriately fit the environment. A machine or language-dependent tool must be customized, and even a portable tool is more often than not "almost" portable.

Probably the most important aid and the first one that should be available is the programmer's reference manual. A comprehensive reference manual containing "what every programmer should know" about the resources available to work with, the environment he must work within, the programming and documentation standards he must comply with, and the guidelines for developing good, standardized code. This manual is the authoritative source defining the ground rules and conditions for developing the software.

Other tools and techniques which are candidates for selection are:

- FORTRAN Structuring preprocessor
- Documentation aids
- Checkout/debugging aids
- Updating aids
- Walk-throughs
- Desk Checking
- Standards Checking Tools
- Execution Analyzers
- Path Analyzers
- Cross Reference Analyzers
- Test Data Generators
- Test Case Selectors
- Report Writers
- Performance Analyzers

Individual tools are examined and their capabilities are presented in Appendix A and Appendix C of the Final Report for this Study. The appropriate point at which to apply these tools and techniques is discussed in Section 3.4.3.

In the on-going development of large systems that take several years to complete, the evaluation and selection of aids should be performed on a periodic basis in order to take advantage of new technologies or to supplement existing technologies with enhancements.

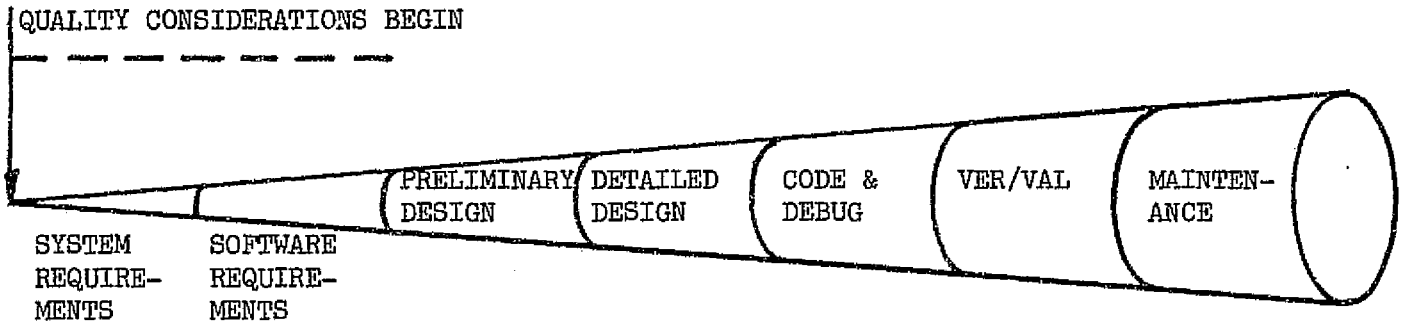
The GSFC computer configuration (the IBM series with TSO capability) coupled with the use of FORTRAN as the universal programming language lends itself to the use of several general purpose tools that are either portable or easily adaptable to the computer center environment. These tools are application independent, therefore scheduling of their installation to meet a development schedule is not a prime consideration.

Other tools that show promise are either designed for a different environment (i.e., language and/or machine) or are application-dependent. These tools should be considered as candidates for development specifically for use at GSFC.

3.5.3 When to Do What

The importance of timing is paramount in the development of software using the built-in quality concept. The decision to use certain techniques and tools at a point that allows errors and faults to be prevented rather than detected must be made very early in the development cycle.

The following set of charts show the points at which various quality considerations should be imposed. The development cycle is shown as the conventional series of steps shown in Figure 3.1. While actual development does not take place in such clearly delineated steps, the phases serve to show the framework upon which decisions can be imposed.



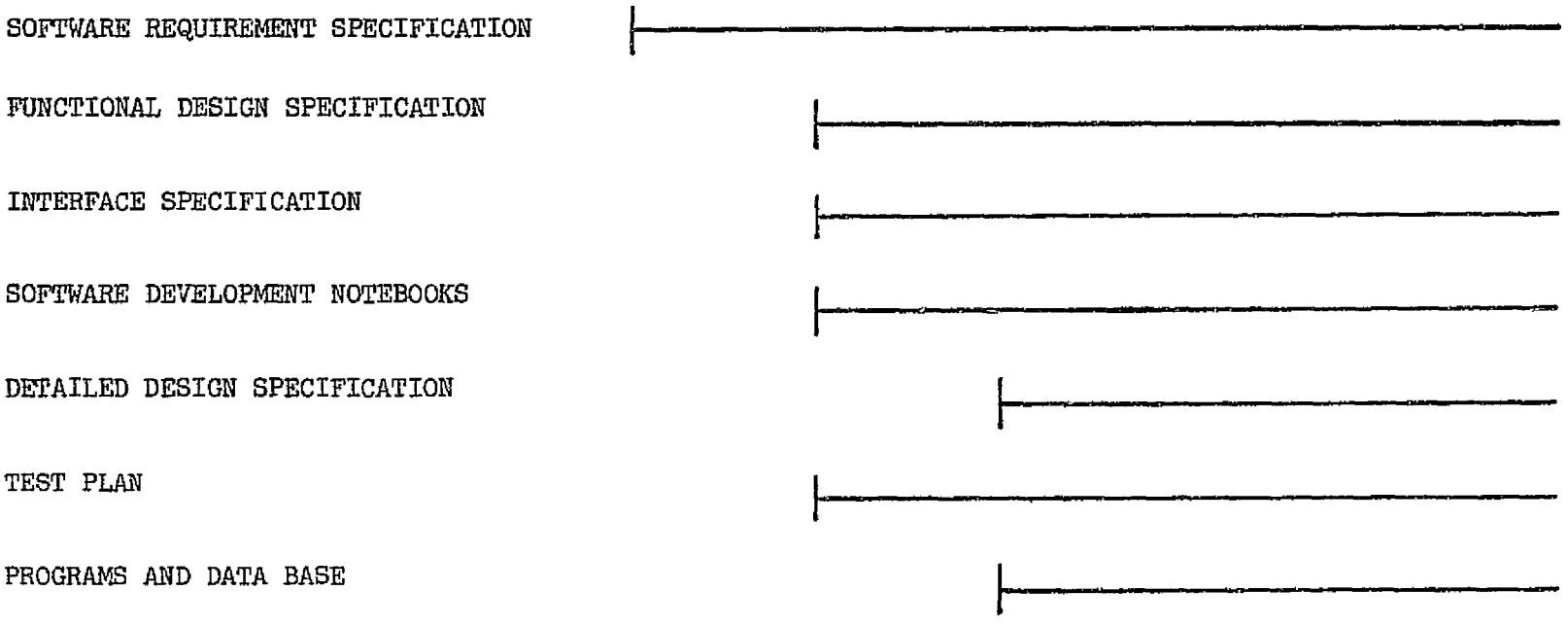
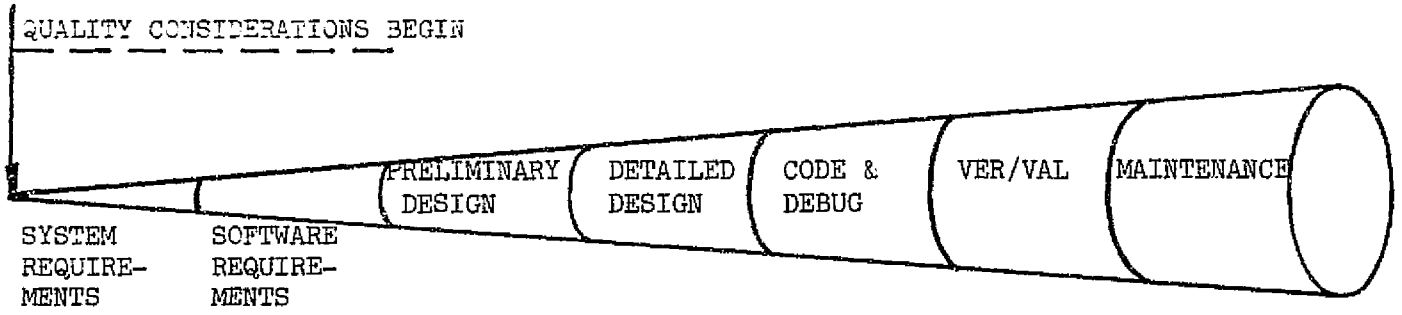


Figure 3.1
EFFECT OF EARLY QUALITY CONSIDERATIONS
CONFIGURATION MANAGEMENT



IDENTIFY PROGRAM MODULES

PLACE DUMMY PROCESSORS UNDER CONTROL

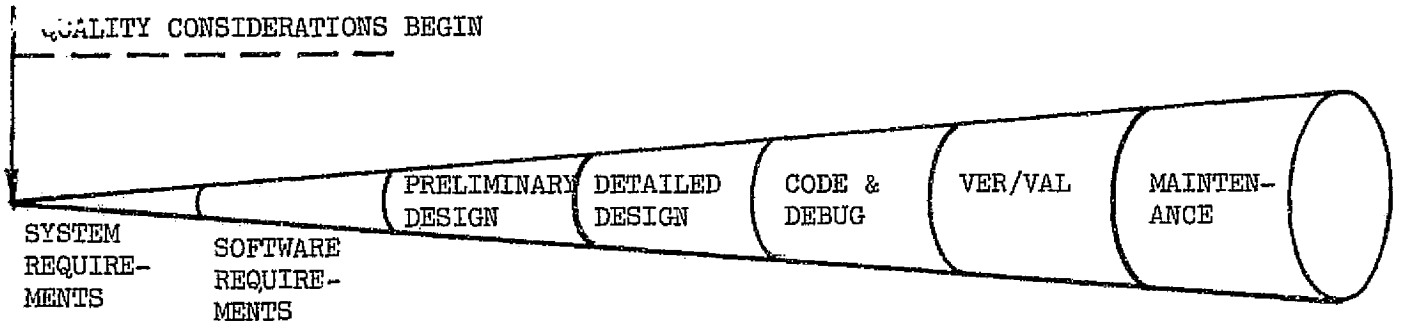
UPDATE DUMMY PROCESSORS WITH CONTROLLED CODE

FORMAL BUILD OF SYSTEM FOR TESTING

RELEASE OF ACCEPTED SYSTEM

UPDATE CONTROLLED CODE

Figure 3.1 (continued)
EFFECT OF EARLY QUALITY CONSIDERATIONS
PROGRAM CONTROL LIBRARY



DOCUMENTATION AIDS

TOPDOWN DEVELOPMENT TECHNIQUE

TEST DATA GENERATOR

PROGRAMMER'S REFERENCE MANUAL

DEBUGGING AIDS

SOFTWARE DEVELOPMENT NOTEBOOKS

UPDATE/COMPILATION PROGRAM

STANDARDS CHECKER

EXECUTION ANALYZER

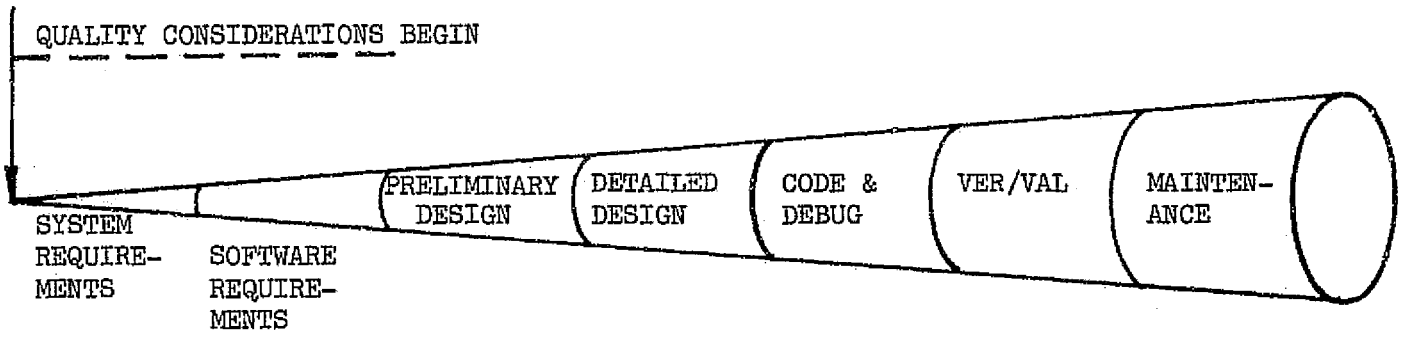
WALK-THROUGHS & DESK CHECKS

AUTOMATED CONFIGURATION MANAGEMENT SYSTEM

SIMULATORS/MODELS

PROGRAM LIBRARY CONTROL

Figure 3.1 (continued)
EFFECT OF EARLY QUALITY CONSIDERATIONS
TOOL AND TECHNIQUE INTRODUCTION



SYSTEM REQUIREMENTS REVIEW (SRR)

SOFTWARE DESIGN REVIEW (SDR)

PRELIMINARY DESIGN REVIEW (PDR)

CRITICAL DESIGN REVIEW (CDR)

CONFIGURATION REVIEW BOARD

PHYSICAL AND FUNCTIONAL CONFIGURATION AUDITS

TEST REVIEW BOARD

DEVELOPMENT TEST REVIEW

INTEGRATION TEST REVIEW

FORMAL QUALIFICATION TEST REVIEW (FQR)

Figure 3.1 (continued)
EFFECT OF EARLY QUALITY CONSIDERATIONS
REVIEWS AND AUDITS

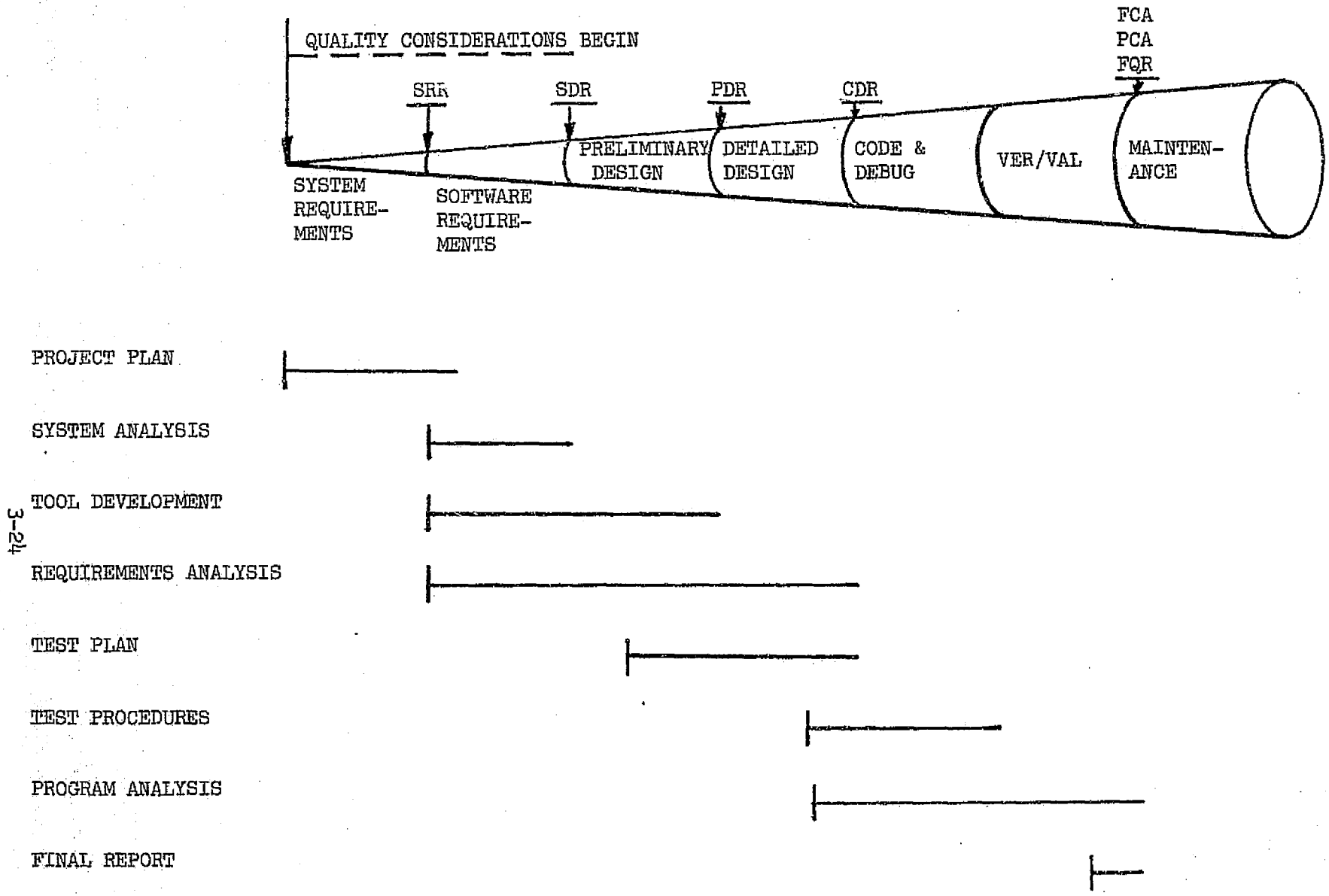


Figure 3.1 (continued)
EFFECT OF EARLY QUALITY CONSIDERATIONS
OVERALL DEVELOPMENT PROCESS

References

1. R. D. Williams. Managing the Development of Reliable Software. International Conference on Reliable Software, Los Angeles, California, April 1975.

Section 4

SOFTWARE VERIFICATION/VALIDATION TECHNIQUES

4.1 GENERAL

The verification and validation of software is the most difficult process of software development, and therefore potentially the most costly. The cost depends largely upon the extent of reliability required of the software. However, the application of some specific tools and techniques can increase the probability of detecting errors, reduce the time required to detect and remove them, and help detect them at the earliest possible point in the development cycle. This can effectively reduce the cost of achieving a required level of reliability. The verification/validation process begins with the analysis and verification of initial requirements and continues throughout the entire development cycle and into the operational state of the system.

This section deals with five aspects of verification/validation:

1. Requirements Analysis and Feedback
2. Code Analysis and Verification
3. Program Validation
4. Program Certification
5. Reliability Determination

The analysis of specifications and the formal and informal reviews of these analyses are discussed in relation to NASA Goddard's current procedures.

Code analysis and verification techniques and tools are available and can, in many instances, be applied directly to the NASA Goddard environment. These include both manual and automated methods of analyzing and verifying the code.

Testing of the software is done in several phases. Each is discussed, with recommendation as to tools and techniques available to enhance the process.

While certification is not a direct requirement of NASA Goddard, its performance is discussed in relation to its positive effect on software reliability.

The need for assessing the reliability of the software required techniques that are still in an experimental state. However, achieving a reasonably accurate determination is possible. The most promising methods of determining reliability are offered for consideration.

4.2 REQUIREMENTS ANALYSIS AND FEEDBACK

The most important aspect of designing systems with built-in quality is the early verification of requirements. Analysis of several large systems shows that as requirements analysis and design time increases, testing time decreases.

In general, a high percentage of errors are attributable to conceptual errors. The early detection of these errors reduces or eliminates their impact in later stages of development.

This section addresses the problem of assuring that the requirements are adequately defined and stated, and that the design reflects the requirements correctly.

This methodology recommends a series of reviews to assure that the intent and the requirements are compatible and that the requestor, the designer the implementer and the tester mutually understand them.

All analysis and review activities should whenever possible be performed by an independent analysis group which may be either an internal or outside-contractor group of analysts.

4.2.1 System Requirements Review (SRR)

The purpose of the System Requirements Review (SRR) is to assure that the system requirements are feasible and that they are complete and unambiguous. The review may include the results of:

- mission analysis
- simulations of the system
- functional flow analysis
- preliminary requirements allocation
- system/cost effectiveness analysis
- trade-off studies
- integrated logistic support analysis
- system interface studies
- program risk analysis
- producibility analysis
- technical performance measurement planning
- integrated test plan
- data management plan
- configuration management plan
- engineering integration plan
- acceptance criteria generation
- system safety definition

Special attention is given to:

- risk factors, their identification and ranking as pointed up in the system/cost effectiveness analysis and technical performance measurement plan analysis, their avoidance/reduction and control

as indicated by analysis of trade-off studies, test planning, hardware proofing, and technical performance measurement.

- significant trade-offs between stated system specification requirements/constraints and resulting engineering requirements/ constraints.

- significant producibility considerations that are visible early in the program, such as manpower loading and hardware availability.

For large systems, SRR's may be conducted for each operational and support subsystem depending on the nature and complexity of the program.

4.2.2 System Design Review (SDR)

The objective of this review is to evaluate the completeness, traceability, correlation, optimization and the risk associated with the proposed system design. It encompasses the total systems requirements, i.e., operations/maintenance/test/computer programs/facilities/personnel/and procedures. A summary review of the items covered in the System Requirements Review that produced the above definitions is included.

The end result of the review is the assurance of a mutual technical understanding of the validity of the system specification and the engineering/cost realism involved in producing the system. The following items are to be achieved in the SDR.

1. Insure that the updated/completed system specification is adequate and cost effective in satisfying validated mission requirements.

2. Insure that the allocated requirements represent a complete and optimal synthesis of the system requirements.

3. Insure that the technical program risks are identified, ranked, avoided, and reduced through:

    a. adequate trade-offs
    b. a responsive test program
    c. subsystem/component hardware proofing
    d. implementation of comprehensive engineering disciplines such as worst case analysis, failure mode and effects analysis, reliability analysis, and standardization.

4. Identify how the final combinations of operations, maintenance, and tests and acceptability requirements have affected overall program concepts.

5. Insure that a technical understanding of the requirements has been reached and technical direction is provided to the implementers.

The SDR re-addresses the items reviewed in the SRR, plus the following items:

1. updated design requirements for operations/maintenance functions.

2. updated operations/maintenance requirements for facilities.

3. updated requirments for operations/maintenance personnel and training.

4. evaluation of

    a. system design feasibility and system/cost effectiveness

    b. capability of the selected hardware/software configuration to meet the requirements of the system specifications

    c. allocated inter- and intra- system interface requirements

    d. specific design concepts that may require development toward advancing the state-of-the-art.

    e. the ability of requirements items to meet overall system requirements and compatibility between requirements items and configuration item interfaces.

    f. reliability trade studies

    g. review of the specification of critical items to assure their traceability/correlation to the validated mission/support requirements.

    h. review of all available test documentation, including subsystem and system test plans to assure that the proposed test program satisfies the test requirements specified in the system specification.

    i. review of computer programming requirements including
        - type of processing, such as on-line processing off-line processing, parallel or multi-processing, multiprogramming, time sharing, etc.
        - a gross description of the size and operating characteristics of all computer programs, including data bases.
        - a description of the requirements for system exercising and identification of functional requirements (exercise configuration, conditions, missions, frequencies, functional simulation, recording and analysis) and identification of major elements to implement the exercising capability.

- identification of programs required throughout the system, such as operational programs, diagnostic programs, test/debug programs, simulation programs, exercise and analysis programs and other support programs.

- identification of computer facility resources needed to support the developing and operational system.

4.2.3 Preliminary Design Review (PDR)

The preliminary design review is a formal technical review of the basic design approach. For large programs, a PDR is conducted for each functionally related group of configuration items. The PDR is the most critical review of the software development review series. This is the point where the conceptual design is accepted and the software system is built upon it. Errors left undetected in the design are often propagated throughout the other phases, causing grief in the later stages.

The items reviewed in a software PDR include:

1. Computer program functional flow.
   This information is completed to the level of flow charting which identifies the allocation of computer program components to functions and depicts the sequence of operation within the system functional flow.

2. Storage Allocation Charts, describing the manner in which available storage is allocated to individual computer programs. Timing, sequencing requirements, and relevant equipment constraints used in determining the allocation are included.

3. Control Functions Description containing a description of the executive control and start/recovery features for the computer program system. It includes the method of initiating system operation and features enabling recovery from system malfunction.

4. Structure and organization of the Data Base identifying data types and characteristics, structure and layout, and allocation of data storage.

5. Standards and conventions to be used in generating and testing the system.

6. Test plans in relation to their ability to demonstrate that the completed software system satisfies the requirements.

7. Configuration identification of major modules.

8. Interface definitions describing the relationships of software system components, for assurance that a particular item does not adversely impact or is not adversely impacted by other system elements.

PDR's are conducted until the software system design is accepted as satisfactory. No detailed design or coding is performed until the preliminary design is complete.

4.2.4 Critical Design Review (CDR)

The purpose of the Critical Design Review is to determine that the detail design of the configuration item under review satisfies the design requirements in the specification for the item, and to establish the exact interface relationships between the configuration item and other related items.

The CDR for each configuration item is conducted prior to the release of the design for production of the software, and the result of each CDR is to commit the design to production.

For computer program configuration items, the CDR is a formal technical review of the item design. The CDR is normally accomplished for the purpose of establishing integrity of computer program design at the level of flow charts or computer program logical design prior to coding and testing. When a given item is a complex aggregate of computer program components, the CDR is accomplished in increments during the development process corresponding to periods at which components or groups of components reach the completion of logical design. For less complex items, the CDR is accomplished at a single review meeting.

The primary product of the CDR is formal identification of specific computer programming documentation which will be released for coding and testing.

Documents to be reviewed include:

- Draft of a complete detail design specification for the computer program configuration item under review.
- Supporting documentation describing results of analyses, testing, etc.
- Documentation of allocated resources for the item
- Test requirements for the item including asserted program properties (reviewed for completeness and technical adequacy).
- Test documentation required to support the test requirements, test procedures in particular.
- Configuration documentation for each item.

4.3 CODE ANALYSIS AND VERIFICATION

Attempts to check code for accuracy and efficiency have taken many forms. Two manual techniques have been found effective in reducing errors when applied systematically. There has also been a proliferation of tools, developed to a large extent on an experimental basis, that are designed to enforce standardization of code and to aid in checking it for inconsistencies, incompleteness, and other structural faults that may cause problems later in the execution of the logic. They also help in determining efficiency of the code in many cases.

Except for the manual technique of "walk throughs", no currently available tools address the function of a program. Some promising steps are now being taken to address function in application-independent tools. One such technique involving the use of an embedded assertion language with accompanying tools is presented in Appendix C.

Tools that are application-dependent and/or environment-dependent must be designed and built as needed.

4.3.1 Manual Techniques

The following two manual code checking techniques should be performed as standard procedure:

a. At the completion of the coding of any module, and prior to submittal for compilation, the application programmer shall: (1) desk-check his module, following the procedures described in 4.3.1.1 until no additional errors are discovered; (2) update the flowchart of the module to reflect any coding modifications; (3) review the module's flowchart with his auxiliary programmer; (4) submit the module for desk-checking by the auxiliary programmer; (5) repeat the above steps (1-4) until no additional errors are discovered; (6) obtain the auxiliary programmer's approval on the module development form (MDF - see 4.3.1.2).

b. Obtain an error-free program compilation

c. Update the program flowchart to reflect the valid compilation.

d. Review the updated flowchart with the auxiliary programmer and obtain his approval on the RDF.

e. Prepare sufficient module development test data, as described in 4.4.1.

f. Submit the module program design language (PDL), description or flowchart to a group walk-through, as defined in section 4.3.1.2.

g. Test the module with the test-data; review the results of each test run with the auxiliary programmer.

h. After development testing has been satisfactorily completed, a public presentation of the code will be conducted (see 4.3.1.2).

4.3.1.1 Desk Checking Procedures

The sample procedure listed below illustrates a method which can be followed in desk-checking a FORTRAN module:

1. Answer the following checklist questions:

   a. Does the commentary block define the purpose, names and definitions of all variables that are transmitted to, and/or from, the routine; and contain version date, programmer's name, references and any special considerations?

   b. Does the commentary block immediately follow the subroutine name?

   c. Are there sufficient comments interspersed throughout the code to explain the general logic flow?

   d. Have embedded assertions been included both as text in the design documentation and as comments in the code for checking:

      data integrity
      entry/exit constraints
      result validity
      local data constraints
      local addressing constraints

   e. Are declaratives in the following order?

      TYPE statements
      DIMENSION statements
      BLANK COMMON statements
      Labelled COMMON statements
      EQUIVALENCE statements
      DATA statements

   f. Are the declaratives blocked so they are easily readable?

   g. Do all floating point variables begin with letters A-H or O-Z?

   h. Do all fixed point variables begin with letters I-N?

   i. Do logical and complex variables begin with letters appropriate to the function of the variable?

   j. Do all variable and subroutine names suggest their function?

   k. Do all variables in a Common block use the same name in every subroutine in which it appears?

l. Are variables passed between subroutines by the use of COMMON rather than by calling parameters?

m. Do all COMMON blocks contain less than seven arguments?

n. Has EQUIVALENCE been used to identify specific locations in COMMON block arrays?

o. Are all RETURN statements, GOTO statements and CALLS and function calls labelled?

p. Is the normal RETURN statement labelled 999?

q. Are all exceptional RETURN statements labelled with a 99x?

r. Are all labels in ascending order?

s. Are all COMMON variables initialized in a BLOCK DATA subroutine, and defined by COMMENT cards?

t. Are the variables in a COMMON block in the following length order?

- COMPLEX*16
- COMPLEX*8
- REAL*8
- REAL*4
- LOGICAL
- INTEGER*2
- INTEGER*2
- LOGICAL*1

u. Are all continuation cards numbered in sequence in column 6?

v. Do parentheses balance? Start from the left with 0 and add 1 for each parenthesis and subtract 1 for each right parenthesis. The count should never become negative. If parentheses balance, the count will end up to 0. However, this does not indicate correct grouping.

w. Do FORMAT statements follow the declaratives and precede the executable code?

x. Are all FORMAT statements labelled in the 99xx range in ascending order?

y. Do the subroutines have less than 100 lines of code?

2. Using the Program Design Language (PDL) description or subroutine flowchart, manually follow the execution sequence of routine logic. This entails:

   a. Preparing sufficient test data to insure that each function within the routine will be exercised at least once;

   b. Manually record each change of program data, variables, counters, and indexes (using the prepared test data to drive the routine);

   c. Verify that the program logic flow accurately reflects the program requirements;

   d. Correct all discovered errors and repeat the above process.

The desk check procedures can be greatly assisted by using a standards checking tool. This type of tool can check for standards violations and flag them for correction. One such tool is described in more detail in Section 4.3.2.1.

4.3.1.2 Walk-throughs

Management shall divide the programming staff into groups of three or more programmer/analysts. Each group constitutes a review group, which will collectively review each group member's programs. There should be two reviews during the development of each program developed by a group member: (1) prior to development test, but after coding is complete; (2) subsequent to development testing. These reviews are conducted to ferret out program logic errors and to insure that the program has been thoroughly tested.

Each review is termed a "walk-through", where the application programmer conducting the review explains his module to the other group members.

These reviews typically take the form of a viewgraph presentation of the modules PDL or flowchart, where the cognizant programmer traces the logic flow (i.e., wlaking the other group members through the module logic). The initial review is informal and made to programmers. The public presentation after development test is formal and is made to the Design Group.

During each review, errors may be detected by the members of the review group. Each error discovered will be recorded by the auxiliary programmer and serve as an action item list for the cognizant programmer. During both reviews, the action items discovered will be recorded in the Module Development Form (MDF) (see Figure 4-1); prior to final approval of the development testing completion for a particular routine, the review group should insure that all action items have been corrected.

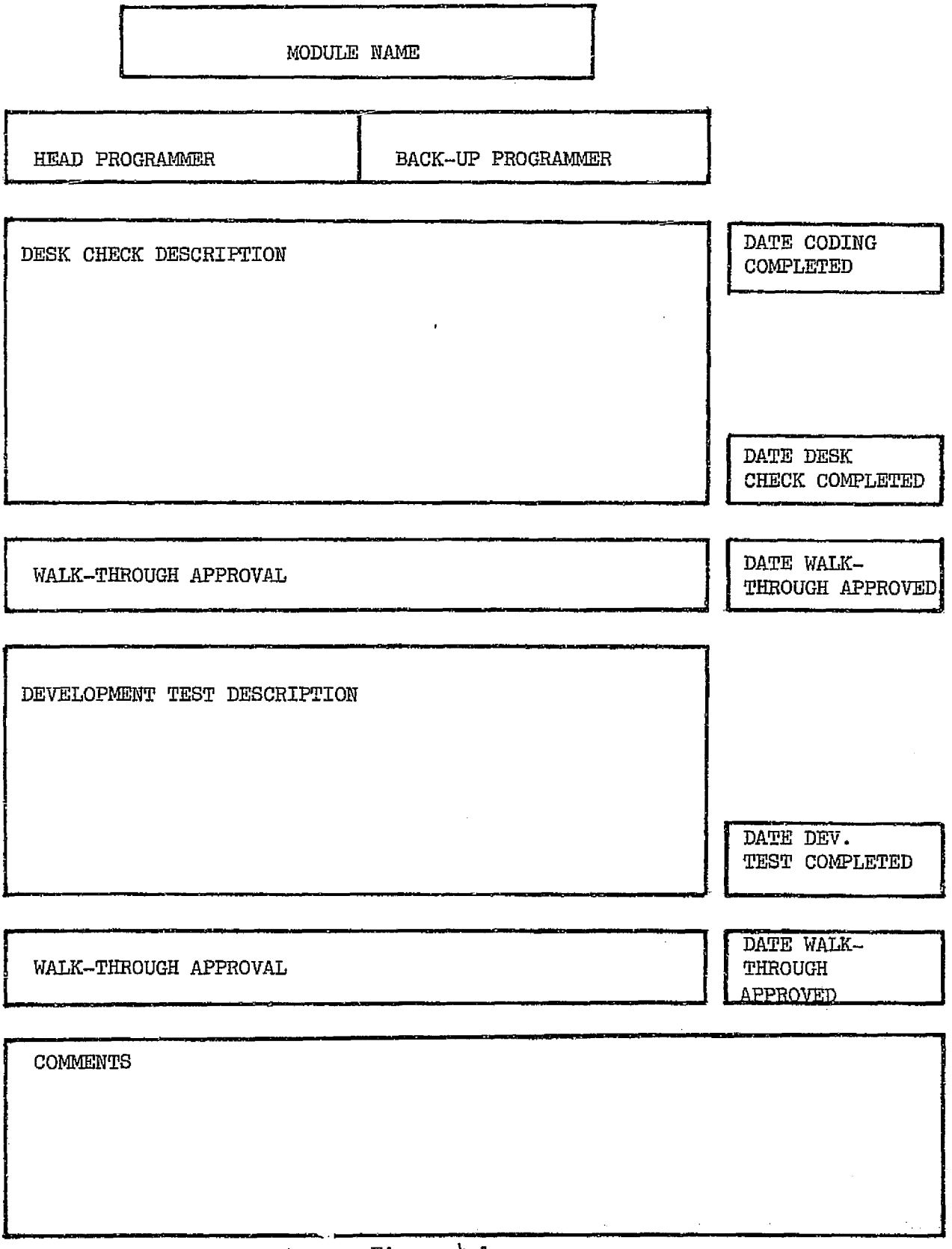


Figure 4-1
SAMPLE MODULE DEVELOPMENT FORM

4.3.2 Automated Techniques

4.3.2.1 Standards Checkers

The use of a standards checking tool can be used to complement the manual desk checking techniques mentioned earlier. Two tools seem worthy of mentioning as examples. The first, PFORT Verifier, Bell Laboratories, Murray Hill, New Jersey is a very useful tool for checking the portability of FORTRAN programs. The second, Standards Auditor, Computer Software Analysts, Inc., Los Angeles, California is a tool which was originally built to check the coding standards for the Army's Site Defense Project being built by MDAC and TRW.

The PFORT Verifier checks a FORTRAN source program for adherence to a portable subset of ANS FORTRAN. Subprogram communication is checked through common and argument lists. Debugging and documentation aids include subprogram cross reference giving type, usage and attributes of each identifier with a list of statements in which it occurs. Also provided is a summary by subprogram listing argument attributes, common blocks used, subprograms called, and the calling subprograms. PFORT has been installed at a number of locations and is available from Bell Labs. Appendix A contains sample output from the PFORT system.

Standards Auditor currently checks 38 coding standards. It has a suppression capability that allows selection of any subset of these 38 standards. Additional standards can be readily added.

It has been found that the most benefits accrue from checking a small core of standards which include statement location, comments and module size. Standards Auditor is marketed as a program product by CSA. A program was supplied to CSA for analysis in connection with this study, however, no output was received for inclusion in this study report.

4.3.2.2 Execution Analyzers

There are many execution analyzers that have been built on both a commercial and experimental basis. Most are designed for FORTRAN code, with a few commercial ones handling COBOL code.

The Boole and Babbage problem program evaluator (PPE) already in use at GSFC, is language independent since it is applied to object code. Tools such as the McDonnell Douglas Program Evaluator and Tester (PET), the CAPEX FORTUNE, and the National Bureau of Standards Analyzer instrument the FORTRAN source program.

PPE is a valuable tool for measuring the performance of an executing program. Its main advantage is that it resides in the same region as the problem program being measured and can readily be applied to programs during production runs. This helps to determine the performance of a program in a real use environment while operating with actual rather than test data. The executing program is not modified in any way, therefore, there is little chance of degrading the program's functional capability, except where a timing function is involved.

The disadvantage inherent in the use of PPE is the difficulty encountered in interpreting the results.

For a total picture of a program's efficiency, PPE is a good tool to use. While other tools can detail the frequency of execution of each statement, timing considerations are difficult to arrive at accurately.

A second tool is recommended to supplement PPE at GSFC. This is the FORTRAN analyzer, PET, produced by McDonnell Douglas.

This tool is designed around a preprocessor/postprocessor organization. The preprocessor inserts software probes into the target code. The postprocessor analyzes the data collected by the probes, and writes several summary reports containing the results.

Run time statistics include:

1. the number and percentage of the total of executable statements, non-executable statements, and comments.

2. the number of and percentage of all potential executable statements that were executed one or more times.

3. the number of and percentage of program branches tested.

4. the number of times each branch was executed. This includes branch counts for logical and arithmetic IF conditions, plus computed and assigned GOTO's branching histories.

5. the number and percentage of subroutine calls that were executed.

6. the number of times each subroutine was called, and the names of those subroutines that were never entered.

7. relative timing for subroutine executions

8. the number of times each executable statement was executed.

9. the minimum and maximum values attained by an assignment statement variable or DO loop parameter.

10. the first and last values attained by an assignment statement variable or DO loop parameter.

The data collected and reported by PET can be used to:

- show areas of high activity during execution of various test cases.
- show untested code
- develop test cases that exercise the entire program

While the application of PET to the code does not prove the correctness of the algorithms, it does allow the observation of the behavior of the algorithms with actual test data. Future plans include the incorporation of an assertion capability that will address the correctness of the algorithms to some extent (see Appendix C).

The chief value of PET is its assistance in deriving adequate test data for development testing. By showing that all of the code was tested using valid test data, the credibility of a program's correct performance is enhanced prior to the final walk through.

Other execution analyzers have been built for FORTRAN on the CDC and UNIVAC equipment, however, none of these tools are available on the IBM 360/370 series computers (see Appendices A and C).

4.3.2.3 Cross Reference Analyzer

As programs increase in size, the problem of naming conflicts increases. This is particularly true when enhancements are made to existing programs.

A cross-reference analyzer creates glossaries verifying the consistency of symbol naming and usage.

JOYCE is a tool produced by McDonnell Douglas which provides cross reference lists for FORTRAN programs. The symbols referenced include the names of any referenced module or functions, any entry points, and all I/O file references. The cross reference lists are also useful for finding typographical errors in coding and for checking a program's logic flow. Sample outputs are contained in Appendix A. As is the case with many of the better tools examined in this survey, JOYCE is currently only operational on CDC machines.

4.3.2.4 Path Analyzers

There have been several attempts to develop a path analyzer that is easy to use and that is helpful in debugging and testing of code. However, the path analyzers that are applied to source statements within a module are awkward to use and require that the user have intimate knowledge of the code and the function for which it was created. Their use will become more significant when they are used as forerunners to test data generators which are still in the experimental stage.

Path analyzers at the subroutine level provide visibility over an entire program in regard to subroutine entries.

The Automated Test Data Generator (ATDG) developed for NASA in Houston by TRW is discussed in Appendix A.

ATDG is a path analyzer that is promising. However, it is written to run on a UNIVAC 1110 computer. It is an interactive tool that requires a high degree of user involvement, and an intimate knowledge of the code. It is complicated to use and since it is used only at specific points in the development of a program, programmers often do not want to take the time to understand how it works. However, work continues in the simplification of its use.

When ATDG is completed, it may be a candidate for conversion to the IBM 360 configuration, particularly since it is already a NASA sponsored product.

Another promising tool currently being developed by MDAC is DISSECT, a symbolic evaluation system, used to analyze programs written in ANSI FORTRAN.

When a program path is executed by running the program on a given input, the correctness of the path for that input can be determined by examining the effects of the calculations carried out by the path. If the path is executed "symbolically" rather than with actual data, it may be possible to use a single execution to illustrate its correctness on a large subset of the input domain rather than on just a single value. Symbolic execution of a program is carried out by given dummy symbolic values rather than actual numeric (string, logical, etc.) values to all or some of the input variables of the program. An expression in the program involving variables with symbolic values is evaluated by substituting the current symbolic values of the variables into the expression. The resulting expression is then simplified algebraically. All operators having only actual as opposed to symbolic operands are evaluated in the normal way. The resulting expression is the symbolic value of the original expression.

The command file is built for a DISSECT analysis of a subject program and is divided into a number of cases. The program is analyzed for each case. The system is used to examine desired program paths.

A program path is a possible flow of control through the program. A path is feasible if at least one element of the program's input domain causes that path to be executed. In general, a complete set of DISSECT cases for a program should "cover" the program in some sense. One approach is to analyze each feasible path (up to some number of iterations of loops). Complex programs having many paths can be divided into segments and analyzed using separate cases.

A great deal of interesting research remains to be done in connection with using the DISSECT system to study techniques for examining program correctness questions. Sample output is contained in Appendix A and an additional description of the system is presented in Appendix C.

4.4 PROGRAM VALIDATION

The systematic validation of a computer program begins with the validation of its design, and in theory ends with the formal qualification test. However, in practice, validation extends into the maintenance phase and ends with the demise of the program.

Therefore, this methodology addresses several kinds of testing that span the development period and continues into the operational state.

It may be argued that development testing is not really a validation process but it is included here as the testing that is the transition from the debugging to formalized validation efforts. Its importance is significant

because it assures that the debugging process is complete before the code is integrated into a system for testing. This reduces the chance of failures caused by structural errors, allowing concentration on the detection of logical errors during testing.

The use of a formal qualification test against an exactly known configuration provides a baseline for the system. Any discrepant behavior of the system is documented. The decisions impacting system acceptance and disposition of discrepancies can be made in a formal manner with adequate and visible justification. The operational test provides a means of measuring the system's real time performance in the operational environment. This testing determines if the uptime requirements are met over a pre-specified period of time.

Baseline tests exercise the critical functions of a system to determine the effect of changes. In particular, they are used to insure that the critical functions have not been degraded. Baseline tests and benchmark tests are considered synonomous in this context.

4.4.1 Development Testing

The purpose of the development test is to provide some assurance that a unit of code, generally a subroutine or group of subroutines that perform a function, works as it was intended. This is accomplished by processing a trivial but realistic test case.

The development test is designed and performed by the programmer who wrote the code. He writes a test plan that describes the function of the code, the data he intends to use to demonstrate that code works properly, and the results he expects from the execution of the test.

The programmer discusses his test plan with his auxiliary programmer and gets his concurrence.

The development test should show that the program function is achieved, and also that all of the code is exercised by the test data, so that an unusual occurrence of data combinations will not result in unhappy surprises later. The use of an execution analyzer such as PET will assist greatly in achieving these goals, by automatically checking program assertions and providing the statistics showing the behavior of the program during execution using candidate test data.

The debugged routine(s) is linked to the basic system, the test case(s) executed, and the results documented. When the intended results are obtained, the program is ready for the formal walk through.

The test plan, test data, and the documented results are placed in the software development notebook. The program is turned over to the validation group for formal validation testing.

When development testing of new programs is complete, the code is placed under release control and entered into the controlled library. If the code consists of modifications to routines already under release control,

the modifications are turned over to release control for entry into the controlled library. The necessary configuration change information is then confirmed by release control.

In either case, once the development tested code has been placed in the controlled library, it can only be changed by CCB approval and corrected following release control procedures. This is true even before the formal build of a system for qualification testing, in order to always preserve the integrity of the system. Since other programmers will be development testing their code using the constantly changing parallel system, it is vital that the configuration be explicitly known at all times. This prevents testing against a system whose configuration is assumed, perhaps incorrectly.

4.4.2 The Development of Test Cases

The test program must be planned early in the development cycle. System requirements must be analyzed for testability before the design of the system can be successfully completed. The formulation of a test plan must be based on a well stated requirements document. If a requirement is explicit, its testability can be readily ascertained. However, implicit requirements must be considered also, and the identification of these requirements and the determination of their testability often requires a great deal of though and conscious effort.

The system test plan has as its goal the evaluation of the performance of the system. Its execution must provide positive answers to the question "Will the system do what it is supposed to do?" Since "what it is supposed to do" is specifically identified in the requirements document, it is important to place boundaries on the test plan to insure that all specified requirements have been implemented, and that no unspecified requirements have been implemented.

The system requirements provide the foundation for the selection of test cases. Ultimately, the composite of all test cases, when successfully executed, verify that the requirements of the system have been satisfied.

The process of test case selection begins with the groupings of requirements by function. An example is shown in Figure 4-2. For a specific requirement, objectives are defined and assertions formulated which explicitly identify all the implications of that requirement that are to be verified. As a result of specifying an objective, each requirement is clarified and any weaknesses and ambiguities can be identified and resolved. This process assures that each requirement is testable.

Test design begins by considering each objective and answering the following questions regarding it:

1. what are the outputs required to evaluate the performance?

2. what are the inputs?

3. how does the data need to be analyzed to verify the objectives?

4. what is the acceptance criteria upon which the pass/fail decision is based?

5. what is the software/hardware configuration required to test the objective?

The answers to these questions form the test requirements which when implemented in a test program, will verify the system performance.

When all of the objectives have been identified, a grouping of the objectives is performed. The criteria for grouping may be predicated upon the commonality of the software/hardware configuration and the system inputs required to verify each objective. A group of objectives define a test case.

At this point, the test cases are grouped together to form scenarios that provide the input for a test (see Figure 4-2).

Test cases are grouped on the basis of the operational or chronological relationships of the inputs. With the scenarios defined, the detailed procedures for the conduct of the test can be written.

4.5 PROGRAM CERTIFICATION

Certification of a system is the last step to be taken before acceptance of the system. The purpose of certification is to provide confidence that the system will work as expected with a specified degree of reliability.

The following definition of certification is given by R. C. White[3], and contains all of the elements that characterize certification.

> "Certification is the act of authoritatively confirming that some set of characteristics are compliant with a particular set of requirements for these characteristics/capabilities.
>
> This act may be further characterized by the following features:
>
> - It is an official authoritative affirmation of the compliancy relation's existence.
> - It is issued by a recognized acceptable authority.
> - It is consequent to an affirmative compliancy decision.
> - It may grant official acceptance.
> - It has such force as to encourage, if not compel, acceptance.
> - It has possible legal efficacy, as determined by the recognized authority and the source of his responsibility for certification.

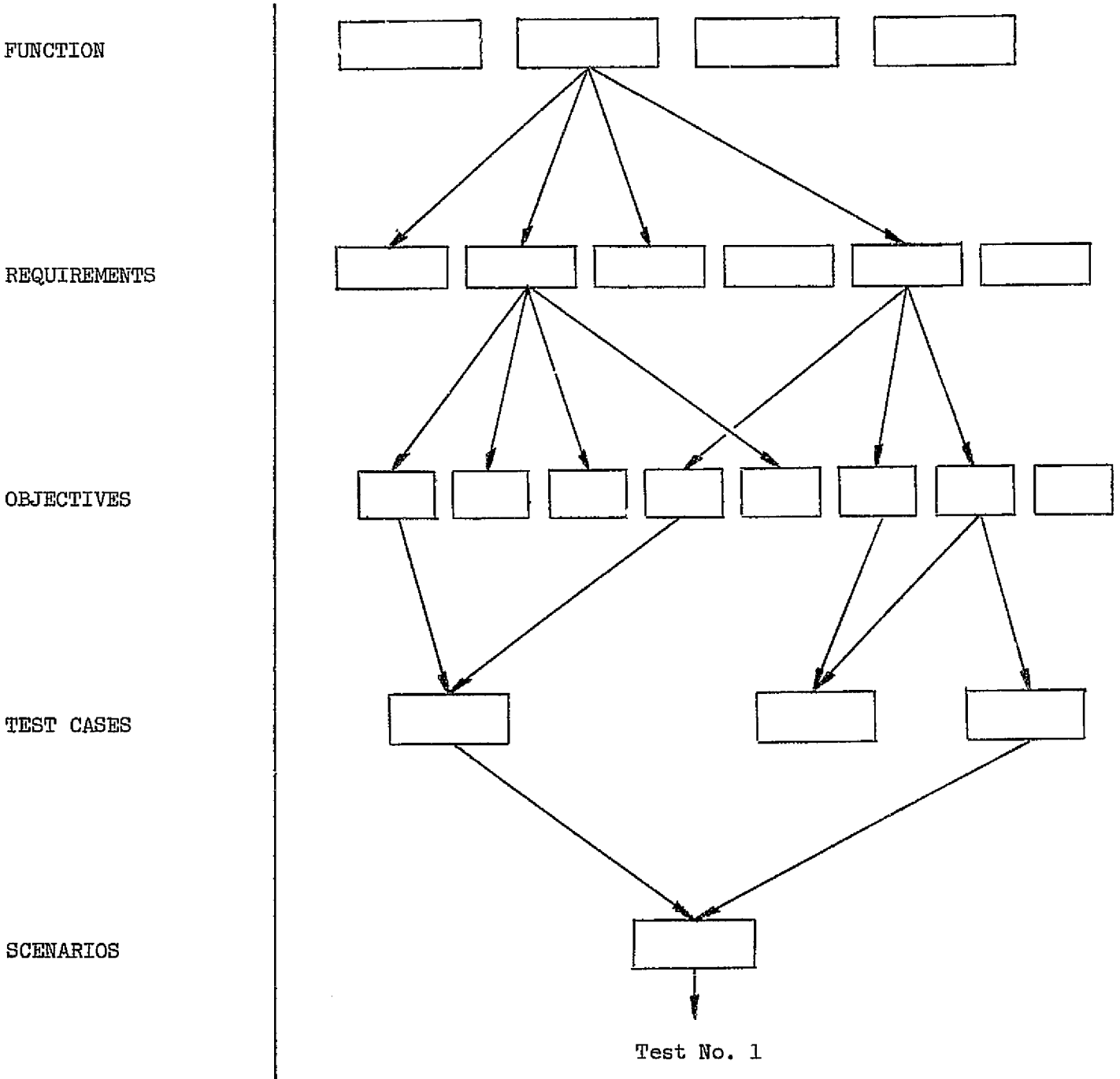


Figure 4-2. Test Design Process

Construed as an act, certification is fundamentally "a single accomplishment complete in itself and essentially unique", in contrast to the extended activity or range of activities, characteristic of compliance determination. Thus, it is an existence-confirming (of the compliancy relation) act, rather than the existence-determination activity of compliancy determination. The latter, to reiterate, is a prerequisite for certification and cannot, therefore, in the interests of consistency, be denoted by the term certification".

The bulleted points in this definition are of particular interest to this methodology, since each represents the result of some quality-producing activity or activities already described in this document.

The act of determining compliancy with the requirement set is the result of on-going analysis of the test data culminating in a final review, the Functional Qualification Review. The analysis and certification should be performed by an independent agency with recognized capabilities to authoritatively confirm compliances. The acceptance of a system can then be based upon explicit recognition of its capabilities and any discrepant behavior that may be deemed non-critical and within the realm of acceptability.

4.5.1 Compliancy Determination

If the technique of defining test objectives for every requirements is used as described in Section 4.4 then it can be determined from the tests results whether or not each characteristic or capability of the system is compliant with a particular requirement of that characteristic or capability.

White maintains that the interpretation of compliancy is binary, i.e., the characteristic or capability either is or is not compliant. It cannot be partially compliant unless the requirement is decomposed into discrete subrequirements which permit separate compliancy determinations. Therefore, the completed system can be certified as compliant if all requirements are satisfied, essentially forcing acceptance. However, if the system fails to conform with one or more requirements, then a negative compliancy decision must be made and alternative action taken. This action may be to return the system for correction of problems, to accept the system without certification, to certify the subset of the system requirements for which conformity was established, or to modify or amend the requirements so that the compliancy decision is affirmative.

The use of the test objective matrix described in Section 4.4 allows the evaluator to check every requirement or subrequirement for compliancy by the inspection, analysis and evaluation of the test results. Since ambiguous or incomplete requirements cause problems in defining test objectives and subsequently in determining compliance, the analysis and feedback of requirements in the initial stages of development not only helps create a cleaner design and implementation of the system, but permits the development of a test plan that will provide a clear indication of requirement compliance.

Compliancy determination is an on-going activity that spans the test phase. The examination of requirements may begin with the beginning of the test plan design and continues throughout. The compliancy determination for each requirement may be made as testing progresses for each subsystem. The final determination of compliancy for the entire system is ba d on the cumulative determinations for every requirement.

The compliancy decision, either affirmative or negative, is a unique one-time process that is made at the Formal Qualification Review following the acceptance test or system test.

4.5.2 Formal Qualification Review

The objective of this review is to verify that the actual performance of the system complies with the requirements as specified. This verification is based upon an analysis and evaluation of the test results.

The Formal Qualification Review examines the results of the analysis and evaluation of the test data with relation to the corresponding requirement set. The end result of the FQR is to determine the disposition of the system (acceptance/rejection) based on the compliancy determination factors. All discrepancies found in the testing are presented, and a decision is made concerning their disposition, generally based on their criticality.

The items presented are:

1. the test objective matrix showing the direct relationship to the requirements and the extent to which the system is shown to comply with the requirements.

2. the test plans and procedures

3. a list of all successful functional tests

4. a test report containing the analysis and evaluation of the test results.

5. discrepancies detected during the acceptance or system test

6. the functional and physical configuration audit data confirming the configuration of the system for which the test data is verified.

7. a list of all completed manuals and handbooks to be used with the system.

8. an affirmative or negative compliancy decision with recommendations for acceptance or rejection.

4.6 RELIABILITY DETERMINATION

The ability to assess the reliability of the software has been a subject of much controversy. Many in the past have considered the attachment of any figure-of-merit to software as an impossible task. Recent research efforts have developed a number of interesting models which have been applied with varying success to software. The key to applying these models involves the availability of error data corresponding to a given piece of software.

Despite the fact that numerous individuals have recognized the need to save error data for some time, remarkably little can be said about software errors. A number of projects have introduced trouble reporting schemes and reams of paper have been generated, however, practically no analysis has been performed on the nature of these errors. Often the information requested on the trouble reports is of little value for such analysis.

Based on the knowledge gained in developing a number of predictive software reliability models, McDonnell Douglas has designed a refined reporting form. After examination of a number of NASA and DOD reporting forms the following sample Software Malfunction Report (SMR) was produced (Figure 4-3).

The software malfunction report (SMR) categorizes malfunctions into six general classes with specific malfunctions as a subset of the general class. The six general classes are:

1. "A" Arithmetic
2. "B" Argument
3. "C" Logical
4. "D" Assignment
5. "E" System
6. "F" Data

Each general class contains varying numbers of specific error types. The specific error types were added to highlight the type of error in each general class. These specific error types are not fixed and they can be deleted or expanded. The specific error types may undergo several major changes until a large enough sample is obtained.

With the availability of the error data by general class and the availability of timing statistics reflecting accumulated development testing times, software reliability models will gain increased accuracy.

Assuming that software error data is gathered, one question still deserves addressing namely: how should the models be applied to the software error data?

ADVANCE INFORMATION SYSTEMS
SOFTWARE MALFUNCTION REPORT

REPORT NO. _____ DATE (1 2 3 4 5 6) ☐☐☐☐☐☐

PROGRAM NAME

PROGRAM I.D. (8 9 10 11 12 13 14 15) ☐☐☐☐☐☐☐☐ MODULE I.D. (16 17 18 19 20 21 22 23) ☐☐☐☐☐☐☐☐ ORIGINATOR

CARD TYPE (7) ☐

STAGE
CHECKOUT (24) ☐ TEST AND EVALUATION (25) ☐ INTEGRATION (26) ☐ USER (27) ☐

SCOPE OF ERROR
SPECIFICATION (28) ☐ DESIGN (29) ☐ CODING (30) ☐ INTEGRATION (31) ☐ OTHER (32) ☐

SEVERITY
HIGH (33) ☐
MEDIUM (34) ☐
LOW (35) ☐

GENERAL ERROR TYPE		SPECIFIC ERROR
ARITHMETIC	36 ☐	1 COMPUTATION; 2 OVERFLOW; 3 SIGN; 4 SCALING; 5 ROUNDING; 6 QUANTITY
ARGUMENT	37 ☐	1 FLAG; 2 CONDITION; 3 LOOP; 4 PARITY; 5 INDEX REGISTER; 6 INSTRUCTION
LOGICAL	38 ☐	1 PROGRAM; 2 PARAMETER; 3 SEQUENCE; 4 COUNTER; 5 INCONSISTENCY; 6 CODE; 7 REQUIREMENT
ASSIGNMENT	39 ☐	1 ADDRESS; 2 ALLOCATION; 3 SUBROUTINE; 4 INTERRUPT; 5 INCOMPATIBLE; 6 ENABLE/DISABLE; 7 MOVE/SORT
SYSTEM	40 ☐	1 INTERMITTENT; 2 LINKAGE; 3 MASKING; 4 SYSTEM STRUCTURE; 5 TIMING; 6 PROCEDURE
DATA	41 ☐	1 STORE/SAVE; 2 CONTROL CARD; 3 FORMAT; 4 CELL; 5 OUTPUT; 6 INPUT

NUMBER OF INSTRUCTIONS (42 43 44 45 46 47) ☐☐☐☐☐☐ NUMBER OF INSTRUCTIONS FOR CORRECTIONS (48 49 50 51 52 53) ☐☐☐☐☐☐ DOES IT USE STANDARD SUBROUTINE YES (54) ☐ NO (55) ☐

IS MODULE CALLABLE BY ITSELF YES (56) ☐ NO (57) ☐ DOES IT USE OTHER MODULE YES (58) ☐ NO (59) ☐

COMPUTER TIME TO PROGRAM HALT (60 61 62 63 64 65) ☐☐☐☐☐☐ TOTAL TIME (66 67 68 69 70 71) ☐☐☐☐☐☐ TOTAL TIME IN STAGE (72 73 74 75 76 77) ☐☐☐☐☐☐

DESCRIPTION AND CAUSE OF MALFUNCTION

Figure 4-3
SOFTWARE MALFUNCTION REPORT

As indicated in the Appendix B section on failure-rate models the initial testing of a program frequently does not correspond to the underlying distributions characteristic of the ultimate, or steady-state testing conditions. While this is so, it is still necessary to record the errors found, whether or not the times of their occurence have any use in direct analysis. As indicated in the selection of the "zero" time for the models, there are good indications that an approximation to the total number of errors in a program can be formed on the basis of the total instruction count. When this count is known, an "apriori" error-per-instruction factor can be applied and to form an approximate error content. This then provides a means of setting a realistic limit to what may be called the initial segment of testing. (In this way,the illustrative application the "zero" time was chosen on the basis of the estimated time of occurence of the half-total error).

In the absence of any apriori estimate of the total error content, for whatever reason it cannot be obtained, it is well to record the times of error occurrence. It is clearly of interest to establish a point in testing where the number of errors per time unit(per CPU second, or per calendar day) decreases. Generally when this occurs any of the models which have been described in Appendix B can be employed to obtain estimates of the total error count on the mean-time-to-failure.

In any of the models, this ultimate convergence insures that they can be applied without regard to the possible transient state of the testing from which the data is obtained. The parameter estimates so obtained are not as good in this case as compared to estimates obtained during the steady state, but they will generally provide good guidance nonetheless.

Appendix A

AUTOMATED VERIFICATION TOOLS

See Volume II

Appendix B

SOFTWARE MODELING

B.1 SUMMARY

Five detection (failure) rate models for the software error process are compared. The de-eutrophication model developed by Jelinski and Moranda has a failure rate which decreases by a constant amount upon the detection and removal of each error. In the geometric de-eutrophication model developed by Moranda, the rate for the "next" error stands in a fixed fractional ratio to its prior value (geometric series). In the geometric-Poisson model, also developed by Moranda, the average number of errors found in a given time period stands in a fixed ratio to the average found in the preceding time period of equal length. The Shooman model assumes the detection rate of the (group of) errors following a debugging interval to be directly proportional to the remnant error content. The Schick-Wolverton model is a variation of the de-eutrophication process in which the detection rate starts at zero after each error and increases linearly until the next error with the slope of the line decreasing after each detection with the magnitude of the slope being directly proportional to the current error content.

All models are based on the assumption that the detection rate depends on the number of errors remaining in the software package.

In this task, for purposes of illustration and comparison, the models are all applied to the same data, consisting of a daily record of the number of errors found in the debugging of a program and the CPU time used. Maximum likelihood estimates of the total error content, mean time to next error, and the degree of testing completeness are developed from a small time segment of the data and estimates are compared where possible.

The estimates of total error content are compared to the total number eventually found. Estimates of the MTTF are developed for local and remote time periods.

Finally, the variances/covariances of three of the models are developed.

B.2 DESCRIPTION OF MODELS

B.2.1 De-Eutrophication Process (Description)

This model, developed by Jelinski and Moranda of MDAC, is based on the assumption that the rate of detection of software anomalies (or errors) is proportional, at any time, to the current error content in the software package, and that all remnant errors are equally likely to occur. This model is illustrated in Figure 1. The initial detection rate is given by $N\phi$, where N is the initial error content, and ϕ is the proportionality constant, but which clearly represents one "error's worth" of contribution to the hazard or detection rate.

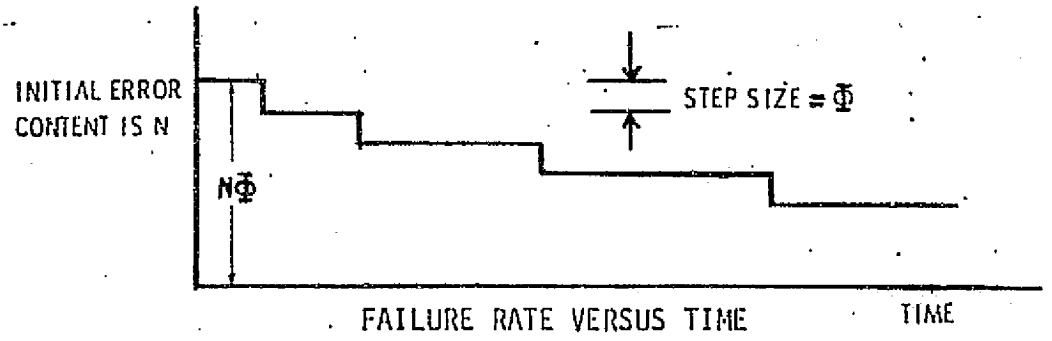


Figure 1. De-Eutrophication Process

A typical realization of such a process is depicted in Figure 2, where errors are indicated by the δ-functions shown, and the time between errors, which is, in reality, random, is purposely indicated here as increasing steadily.

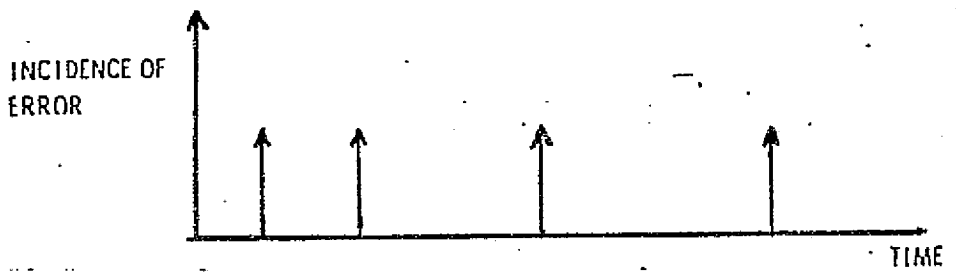


Figure 2. Typical Realization of the De-Eutrophication Process

The data for analysis consists of the sequence of times between errors: $X_1, X_2, \ldots, X_n$. The development of maximum likelihood estimates for the two parameters shown explicitly in Figure 1 is made in Section B.6.1.The essential facts involved in this development are: the uniform or constant conditional failure rate for the ith error implies an exponential distribution for the associated time, X_i, with parameter, $(N-i+1)\phi$; and the X_i's are statistically independent.

The application of the maximum likelihood technique produces the two equations:

$$\sum_{i=1}^{n} \frac{1}{N-(i-1)} = \frac{n}{\frac{N-1}{T} \sum_{i=1} (i-1)X_i} \qquad (1)$$

and

$$\phi = \frac{n}{NT - \sum_{i=1} (i-1)X_i} \qquad (2)$$

where T is the total time $\sum_{i=1} X_i$

Applications of this model have been made to data sets obtained during the development of two large-scale real-time systems; one, the Navy Tactical Data System and, the other an Apollo-related software package. These were reported first in the original paper (Reference 1); subsequently, updated information was obtained. This new data permitted a comparison between the predictions, which had previously been made and the realized data, in the form of Trouble Reports generated during the development of the Naval Tactical Data System during the forecast time period. The comparisons (three modules) are contained in a second report [Reference 2]. Those comparisons showed a remarkable consistency between the predictions and the realizations. In those applications, time was measured in units of days (calendar).

B.2.2 Geometric De-Eutrophication Process (Description)

A variation of the de-eutrophication process has been found useful. In this form the detection rate decreases in a geometric progression on the occurrence of each individual error; the times between errors are random instead of fixed, the errors are treated individually instead of by groups. This is shown in Figure 3.

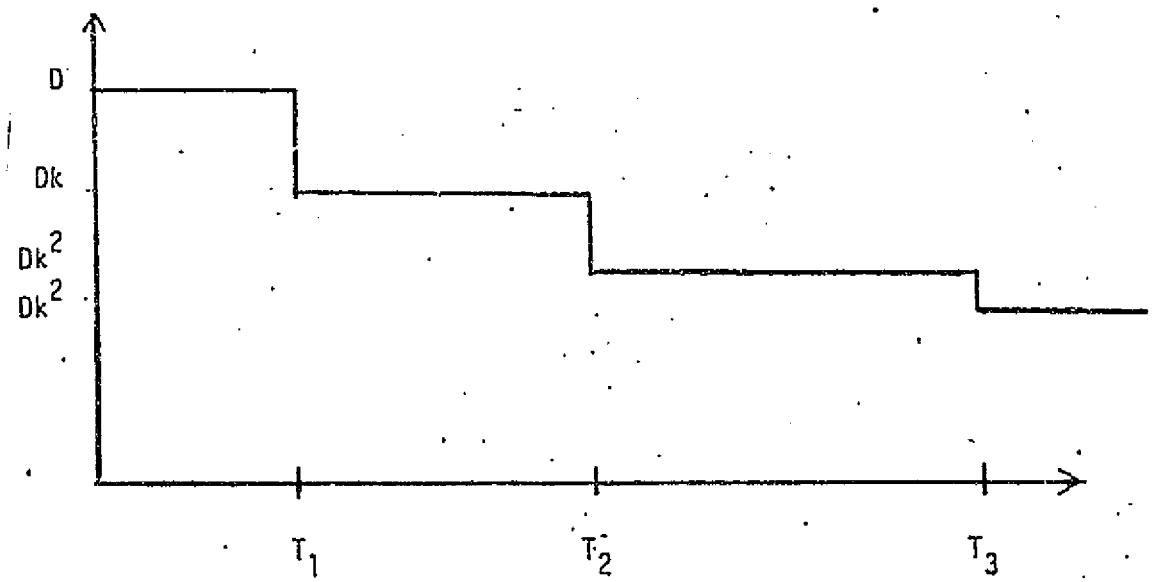


Figure 3. Geometric De-Eutrophication Process

The mathematical analysis for this model is given in Reference 2. The development parallels that used in the original paper: the X_i are exponentially distributed with parameter Dk^{i-1}, the observations are independent and the likelihood function is therefore the product of exponentials. Maximizing the logarithm of the likelihood produces the two equations:

$$\frac{\sum ik^i X_i}{\sum k^i X_i} = \frac{n+1}{2} \qquad (3)$$

and

$$D = \frac{n}{\sum k^{i-1} X_i} \quad , \qquad (4)$$

where k and D are described by Figure (3) and n is the number of intervals used. All sums are over the range 1 to n.

B.2.3 Geometric Poisson Model (Description)

The Geometric-Poisson Model is described by Moranda in Reference 2. The model is shown in Figure 4. As indicated, the data is assumed to be reported only periodically (by week or month). The detection rate is shown to decrease in a geometric progression; each time interval, T, has a rate which is a fraction k times the previous interval's rate ($0<k<1$), and represents the Poisson parameter for the initial collection interval.

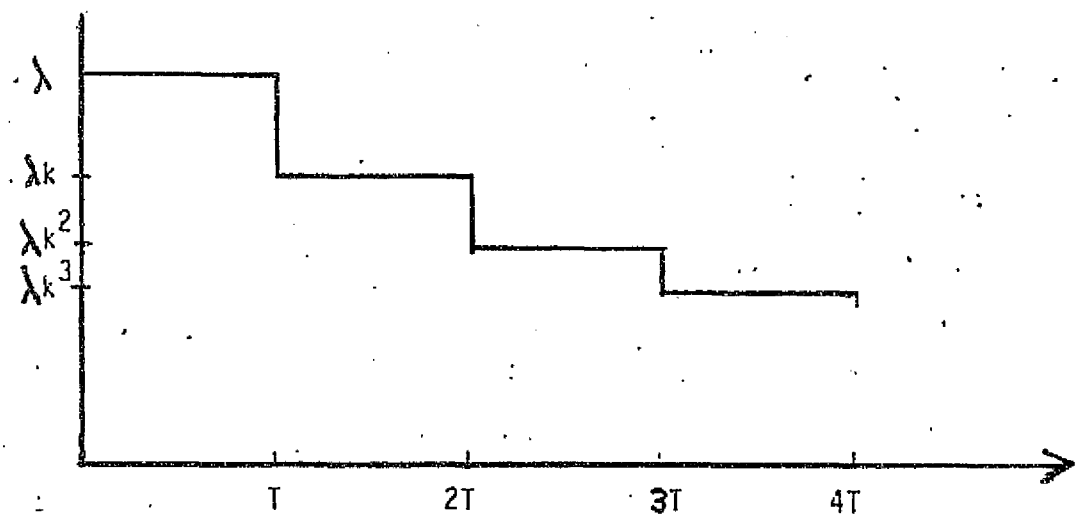


Figure 4. Geometric-Poisson Model

B.2.4 Shooman Model (Description and Critique)

The model described by Shooman initially in November 1971 was presented in improved form in January 1972 in a paper jointly authored by J. C. Dickson, J. L. Hesse, A. C. Kuentz and M. Shooman [References 3 and 4]. The same model and results were described in 1973 and 1975 by Shooman [References 4 and 5].

In the Shooman development, the model is discussed in terms of the factor

$$\gamma = K[\frac{E_T}{I_T} - \varepsilon_C(\tau)]r_p \qquad (5)$$

where: K is a constant "which can be estimated by the ratio of the number of catastrophic errors detected to the total numbers of errors detected"; E_T is the total number of errors; I_T is the total number of instructions, and $\varepsilon_C(\tau)$ is the "normalized" number of errors which have been corrected up to the (total) debugging time τ; and r_p is the instruction processing rate.

As will be later shown, there are several subtle points which must be resolved before the Shooman model can be obtained. It is sufficient here to note that the instruction count I_T does not enter into the problem since the "normalization" required to form $\varepsilon_C(\tau)$ eliminates it. The parameter τ, which represents debugging time, does not enter the analysis, nor do r_p occur as a separate factor, it is obviously inseparable from the factor K.

Thus, in its essence, the Shooman model sets up a direct proportion between the detection rate γ and the current error content.

In Shooman's original paper (Reference 3) when he sets out to find the unknowns, E_T and K, the "constant error rate model" is employed, the assumption being that the

$$\varepsilon_C(\tau) = \rho_o\tau$$

To quote Shooman, the value of ρ_o is "evaluated from previous data". In the only illustrative computations which he carries out, the value of ρ_o is obtained by dividing the area under a triangular-shaped plot of error rate versus time, by the total test time.

Shooman then uses the total number of changes (the area under the plot of error rate versus time) which were observed during the entire test period for the value of E_T. Coupling this with his assumption of a constant error rate, it would seem that the model is complete, but has no predictive potential. Shooman uses it to compute the MTTF versus the debugging time, and illustrates the eventual unlimited magnitude for the MTTF. Since the MTTF is the reciprocal of the factor γ in Eq. 5, and since as τ increases, $\varepsilon_C(\tau)$ increases, the factor γ decreases, and the reciprocal (the MTTF) increases (without limit). This is not, by any means, surprising, since the $\varepsilon_C(\tau)$ tends to approach E_T/I_T, so the denominator tends to zero.

In the second presentation jointly written by J. C. Dickson, J. L. Hesse, A. C. Kientz and M. Shooman, a much improved discussion is made [Reference 4]. In 1973 and 1975 Shooman essentially discussed the same model in the same form given in the original paper [Reference 5 and 6]. He does however distinguish between a microscopic approach, in which individual errors are studied, and the macroscopic approach (in which class his model falls) which treats bugs which are "lumped and treated equally".

In the analysis of his macroscopic model he determines the unknowns K' (the product of K and r_p in his original formulation) and E_T by solving the equations

$$\frac{H_1}{x_{s_1}} = \frac{1}{K'\left[\frac{E_T}{I_T} - \varepsilon_c(\tau_1)\right]} \tag{6}$$

$$\frac{H_2}{x_{s_2}} = \frac{1}{K'\left[\frac{E_T}{I_T} - \varepsilon_c(\tau_2)\right]} \tag{7}$$

where τ_1 and τ_2 are two debugging times with $\tau_1 < \tau_2$ and $\varepsilon_c(\tau_1) < \varepsilon_c(\tau_2)$; x_{s_1} and x_{s_2} are the number of software failures "found during total activation times" H_1, and H_2, respectively.

Each of these two expressions equate a "steady-state" MTTF, on the left side, obtained by observing the error rate over a post-debugging interval (of duration τ) and the factor $1/\gamma$ (of Equation 5), which is the MTTF at the end of the corresponding debugging time. This is referred to as the method of moments.

From the solution which is obtained:

$$E_T = \frac{I_T\left[\lambda_{s_2}\cdot\lambda_{s_1}^{-1}\varepsilon_c(\tau_1) - \varepsilon_c(\tau_2)\right]}{\left[\lambda_{s_2}\lambda_{s_1}^{-1} - 1\right]}$$

with $\lambda_{s_i} = H_i / X_{s_i}$

it is clear that the x_{s1} and x_{s2} represent the number of failures during tests made after τ_1 and τ_2 units of debugging time and the H_1 and H_2 refer to the "locally" cumulative number of hours after the debugging times τ_1 and τ_2 and not the total activation times. This is an important clarification and changes the focus of question of independence of measurements. If these quantities were cumulative from the beginning (instead of locally) then the x_s's are highly correlated, but if they are only locally cumulative, then it can be seen that the independence question is now at the microscopic level; the new question is whether the individual measurements which make up any particular x_s are independent.

This couples with another question and there is a single answer to the two. The other question is: do the individual error-separation-times all estimate the MTTF in an unbiased way? The answer to both is the same: if there are a large number of incipient errors in the package, then the process indeed, resembles the "infinite number of failure-makers" of hardware reliability, and, for a short period after τ_1 (or τ_2), the individual measures of time-between-errors are independent (the coin-flipping analogy) and are unbiased estimates of MTTF (τ). Clearly, since the basic assumption is that the MTTF is (inversely) proportional to the number of residual errors, the number allowed for each x_s must be small, for only the first error found is strictly an unbiased estimate of the MTTF.

The x_{s1} and x_{s2} are "the number of software failures" and as defined by Shooman are the number of unsuccessful runs (not errors) caused by a software "failure". λ_{s1} and λ_{s2} are defined by formula above and clearly represent the rate of failure, and not the rate of error-occurrence. Thus, at best, Shooman is equating two different kinds of failure rate: on the left his (approximate) rate is the number of failures per unit of operating time, while on the right side the rate is the formula-derived number of errors per unit of operating time.

It can be argued that E_T is not the number of errors in the program but some kind of measure of the number of incipient failures. Countering this, however, is the fact that in Shooman's formulation E_T is divided by the total number of instructions I_T, implying that they were thought of strictly in terms of coding errors. (In the applications which were made by Jelinski and Moranda, it was necessary, because of a lack of fine-grained data, to analyze on the basis of "trouble days" instead of on error count, but there was not any consideration of program size in that analysis). [Reference 1].

As an important comment in this respect, it is clear from Equation 7 that the number of instructions I_T is only a nuisance since it is taken out by the "normalization" of $\varepsilon_c(\tau_1)$ and $\varepsilon_c(\tau_2)$, which is required to produce numerical values.

B.2.5 Schick-Wolverton Model (Description and Critique)

George J. Schick and R. W. Wolverton, in September 1972, at Hamburg, Germany, presented a paper in which the de-eutrophication model was described along with a new model which uses the same notation but which describes an entirely different failure rate.

The rationale of the de-eutrophication model, to quote from the original paper, is: "the failure rate at any time is assumed to be proportional to the current error content of the tested program; the initial error content is then denoted by N and the proportionality constant is denoted by ϕ, the failure rate drops to $(N-1)\phi$ after the first error is detected and so forth."

Schick and Wolverton make the comment in their paper that: "there does seem to be an inconsistency by admitting a decreasing failure rate yet at the same time assuming (rather than deriving) an exponential model". Apparently, they interpret the failure rate of the Jelinski-Moranda model as applying to a single error rather than to the sequence of a number of errors. This interpretation cannot be supported by either the analysis of Jelinski-Moranda in their paper or, by Schick-Wolverton's own analysis of their own model as described in their paper.

To make the point more clearly, an examination of the defining equation is useful. In the model, the detection rate is given by

$$Z(t_i) = \phi\,[N-(i-1)]t_i$$

where t_i is "the cumulative time to the occurrence of the ith error". Under that interpretation the variables employed in the likelihood function are not independent. On the other hand if t_i represents the time past the occurrence of the (i-1)st error, then the t_i represents the same measurement as the X_i of the de-eutrophication model; this is the interpretation that agrees with their analysis.

In a purely formal way the likelihood equations for solution are:

$$\frac{2n}{\phi} = \sum_{i=1}^{n} [N-(i-1)]t_i^2 \qquad (8)$$

and

$$\sum_{i=1}^{n} \frac{1}{N-(i-1)} = \frac{\phi}{2} \sum_{i-1}^{n} t_i^2 \qquad (9)$$

where the symbols are the same as those used in Equation 2 (with $X_i=t_i$).

B.3 DATA AND ADJUSTMENTS

Data which is relevant to software errors was obtained recently from W. L. Wagoner [Reference 8]. This data, although not in a form which is ideal for analysis by the three MDAC models since it consists of grouped error counts, was analyzed to obtain estimates of the error content. The unique feature of the data is that the reference unit was CPU time (in seconds).

By adjusting the data in the ways subsequently described it was possible to obtain estimates for all MDAC-models. The results produced estimates which were consistent among the models and accurately predicted the error count which was eventually achieved on the basis of a very short interval of data.

Because of the grouping of the data it is very easy to obtain estimates from the Shooman formulation in either of two interpretations.

In order to complete the comparison, the Schick-Wolverton model is also employed in a formal way.

The data consists of a record of the errors which occurred during the debugging of a data-reduction program (called the F11-D Program) consisting of "approximately 3-4 thousand" Fortran statements. The data is reproduced in part in Table I. The important feature of this is that CPU time is available as a unit.

The three MDAC models describe failure-rates (or Poisson parameters) which decrease with time. The first clear cut evidence of a decreasing failure (found by dividing the errors detected by the CPU time on a daily basis) starts on 1/19 after 5.24 units of CPU time has elapsed. The ratio on 1/16 is 1.54 while on 1/18 it is 10.06 errors per unit of CPU time, while on the next three data-days the ratios are 2.04, 1.31, and 1.10.

This corresponds qualitatively with other experience which has been gained on other data. There is usually a startup effect which is evident. There are fairly clear reasons why this should be so when calendar time is the unit: early in testing there may not be a sufficient number of "working parts" of the package to obtain significant error counts; as time goes on these parts produce in total an increasing error count; finally the assumptions of the models may be met.

Although the same reasons do not necessarily apply to data based on CPU time, some of the general effects seem to be indicated by the data.

It should be said, however, that experience also shows that the two de-eutrophication models can be applied to any initial segment of data. The estimates for the error content will be initially very high (infinite for a constant error rate) but will settle down to give good estimates even though the first part of the data is not being well-modeled.

Table I

DATA ON F11-D PROGRAM

Date	Errors Detected	Cum Error	CPU Time	Cum. CPU Time
1/12	8	8	0.5	.5
1/15	7	15	0.6	1.1
1/16	1	16	0.65	1.75
1/17	8	24	1.90	3.65
1/18	16	40	1.59	5.24
1/19	18	58	8.83	14.07
1/22	13	71	9.94	24.01
1/23	8	79	7.25	31.26
1/24	9	88	8.34	39.60
1/25	2	90	3.86	43.46
1/26	6	96	13.11	56.57
1/27	3	99	34.15	90.72
1/29	3	102	82.7	173.4
1/30	2	104	1.10	174.5
1/31	3	107	51.59	226.11

On the other hand, in order to apply the Geometric-Poisson Model, much greater care has to be taken, since there is much less data which can be employed for the estimates of the two unknowns.

The above reasons are all good reasons for choosing the zero time of the analysis to be the cumulative CPU time at the end of 1/18. But the way in which that time was initially chosen is entirely different. It was chosen initially by applying a universal "Programmers Poisson Parameter" of 1 error per 50 lines of instruction. In order to eliminate the startup effects on a program with 4000 instructions and an estimated 80 errors, the zero time chosen corresponded to the half-way error (40).

While the particular value chosen does not seem to cause any concern, the use of the factor of 1/50 to obtain the a priori error content has caused controversy. As originally stated the rule-of-thumb is that there are (on average) two errors per 100 instructions. This factor has been observed by F. Akiyama on nine fairly large programs. [Reference 9] It also has been noted by B. W. Boehm of TRW in a presentation at the 1974 AFIPS Conference: data from T. A. Thayer, et.al., taken from tests on five large scale programs showed a remarkably consistent rate (22×10^{-3}).[Reference 10]

Boehm in a personal communication, stresses the important fact that the constant he reports is the ratio of errors (program bugs) to the number of source instructions (vis a vis machine or object instructions). But, fortuitously or otherwise, this is exactly the way the figure was employed above in estimating the half-error point (4000 Fortran instructions times 1/50).

Excepting the adjustment for the zero, the data is used as it stands for the two de-eutrophication models. In order to apply the data to the Geometric-Poisson Model, it is necessary to further adjust it. Time intervals of equal size and the number of errors per interval are required for this model. The choice for the length of the intervals is arbitrary (for illustrative purposes); however, five of the six daily CPU times in the time span 1/19 through 1/26 are about 10 seconds in length, so that it is a convenient interval size for comparisons among the models.

In order to apply the data to this interval size, interpolations of cumulative error versus cumulative time are required. As with the previous analyses, and for the reasons given there, the zero time for data corresponds to a cumulative CPU time of 5.24. The data after interpolation and adjustment to the "new" zero, is shown in the first two columns of Table II.

B.4 ESTIMATION OF PARAMETERS

B.4.1 De-Eutrophication Analysis

In the application of the de-eutrophication process the times between successive errors form the primary data. In the present case the data are not recorded in that way. As an expediency the times within any given interval are put equal to one another and have a value equal to the quotient of the CPU time used on a given date, and the number of errors found.

Table II

Adjusted F11-D Data

Interval CPU Time	Number of Errors	Fitted
0-10	19.53	20.35
10-20	12.83	13.75
20-30	10.93	9.28
30-40	7.52	6.27
40-50	4.58	4.23
50-60	1.37	2.86

The analysis is performed in this way on the errors recorded during two days (1/19 and 1/22). Thus, the required data are:

$$X_1 = X_2 = \ldots = X_{18} = .4906$$

and

$$X_{19} = X_{20} = \ldots = X_{31} = .7646$$

This data produces the numerically substituted version of Equation 1 [Reference 1].

$$\sum_{i=1}^{31} \frac{1}{N-(i-1)} = \frac{31}{N-16.7076}$$

which produces an estimated residual error of 63.4 which, together with the 40 "startup" errors, produces an estimate of 103.4 for the total error count.

The estimate for ϕ is obtained by substitution into Equation 2, of Reference 2 and has the value 0.035.

If the same analysis is employed on the data for the three consecutive working days 1/19, 1/22, 1/23, there are 8 additional errors and the time between them is taken to be equal. The additional data is:

$$X_{32} = X_{33} = \ldots X_{39} = .9065$$

The numerical equation to be solved is:

$$\sum_{i=1}^{39} \frac{1}{N-(i-1)} = \frac{39}{N-21.66513}$$

which produces an approximate solution of N=68.8, corresponding to an estimated total error content of 108.8.

The estimate of ϕ in the extended data case, can be computed to yield the value, .032.

The above estimates which are based on just two (or three) day's data yield estimates which "turn out" to be quite good. The program after running seven additional days had uncovered a total of 107 errors. While it is very unlikely that all errors have been found, the time spacing of the latest errors recorded is very long, and the program has a "practical" error content of somewhere between 110 and 115. But it should be pointed out that the data employed in the prediction is only 1/10 (or 1/8) of the total observation time, making the result even more noteworthy.

B.4.2 Geometric De-Eutrophication Analysis

The same data used in the preceding analysis is employed with Equations 8 and 9 to produce estimates of k and D [Reference 2].

Using the data for the first two days (1/19 and 1/22) produces estimates; k = .9735 and D = 2.998.

It is clear that this model cannot be used to estimate the total number of errors; however, it is possible to determine the level of "purity" after n observed errors by evaluating k^n.

The estimated degree of "purification" after n errors have been detected, is given, generally, by the ratio $\frac{R_o - R_f}{R_o}$, where R_o and R_f are the initial and final rates. In this particular case, it is $1-k^n$. For n=31 the degree of purification is 56.4%. The corresponding degree for the de-eutrophication model is given, in general, by the ratio n/N, and for the same data employed earlier this is 31/60.3 or about 51.4% (under a different interpretation, where the initial 40 errors are included in both numerator and denominator, the ratio 71/100.3 = .708, could be used; however, the former figure is clearly the proper one for comparison of the estimates of the two models as they are applied here).

B.4.3 Geometric Poisson Model

As a final analysis of the same data, the Geometric-Poisson Model is employed. The data required has been described in the preceding section and consists of the entries in column 2 of Table II. Using these as the n_i, and substituting into Equations 11 and 12 of Reference 2 produces the polynomial equation:

$$2.4433k^7 - 3.1320k^6 + 1.6887k - k = 0,$$

which has a root k = .6756.

The value of the Poisson parameter is found to be =20.348.

$$= 20.348,$$

Use of these two parameters, produces the third column (labeled "Fitted") of Table II.

By extrapolation (summing the infinite geometric series) the projected total error count is 62.73, (or with the 40, which were "banked", a total of 102.73). This appears at first to be only a fair estimate, when it is compared with an observed total of (at least) 67. But it is important to note once again that the last time-point used in the analysis corresponds to a modified CPU time of 60 seconds, while the 63rd (or 103rd) error occurs after about 168 seconds of (modified) CPU time. With this scale of reference, the estimate is remarkably accurate. The model data can be used to generate forecasts for each time period: for the 60-70, 70-80, 80-90, intervals, they are 1.93, 1.31, and .88, respectively; while the observed counts obtained by interpolation of the actual data are .88, .88, and .64, respectively.

B.4.4 Shooman Model

The Shooman Model does not require a decreasing failure rate and the choice of the time τ_1 is arbitrary. However, in order to achieve some compatibility with the two analyses made with the de-eutrophication models, the time τ_1 is chosen to correspond to the fiducial time, 5.24 CPU-seconds. Choice of the time τ_2 is somewhat open, but to test the model thoroughly, several choices for τ_2 are made, each of which will give an estimate of the error content.

Column 3 of Table I lists directly $\varepsilon_c(\tau)$. For the quantities x_{s_i} and H_i, we employ the narrower, and more proper, interpretation that they relate only to a short time segment subsequent to τ_1 and τ_2. Thus associated with τ_1, are the quantities $x_{s_1} = 18$, and $H_1 = 8.83$, obtained from columns 2 and 4 respectively.

Case I:

$$\tau_1 = 5.24, \quad \tau_2 = 14.07 \ (\approx 3 \text{ times } \tau_1)$$

$$\varepsilon_c(\tau_1)=40, \qquad \varepsilon_c(\tau_2)=58$$

$$x_{s_1} = 18 \qquad x_{s_2} = 13$$

$$H_1 = 8.83 \qquad H_2 = 9.94$$

Thus

$$\lambda_{s_1} = 2.039, \qquad \lambda_{s_2} = 1.307$$

and substituting

$$E_T = \frac{25.663-58}{1-.6415} = \frac{32.337}{.3584} = 90.22$$

This compares to 107 errors found.

Case II:

$$\tau_1 = 5.24 \qquad \tau_2 = 24.01$$

$$\varepsilon(\tau_1)=40 \qquad \varepsilon(\tau_2) = 71$$

$$x_{s_1} = 18 \qquad x_{s_2} = 8$$

$$H_1 = 8.83 \qquad H_2 = 7.25$$

$$\lambda_{s_1} = 2.039 \qquad \lambda_{s_2} = 1.103$$

$$E_T = \frac{(.541)\ 40 - 71}{.541 - 1}$$

$$= 107.54$$

This is almost exactly the number found when the same data was used in the de-eutrophication model, and agrees with the observed number of errors quite well.

Case III:

$\tau_1 = 5.24$ $\qquad$ $\tau_2 = 31.26$

$\varepsilon_c(\tau_1) = 40$ $\qquad$ $\varepsilon_c(\tau_2) = 79$

$x_{s_1} = 18$ $\qquad$ $x_{s_2} = 9$

$H_1 = 8.83$ $\qquad$ $H_2 = 8.34$

Thus

$\lambda_{s_1} = 2.039$ $\qquad$ $\lambda_{s_2} = 1.079$

$E_T = 122.8$

Case IV:

$\tau_1 = 5.24$ $\qquad$ $\tau_2 = 39.60$

$\varepsilon_c(\tau_1) = 40$ $\qquad$ $\varepsilon_c(\tau_2) = 88$

$x_{s_i} = 18$ $\qquad$ $x_{s_2} = 2$

$H_1 = 8.83$ $\qquad$ $H_2 = 3.86$

Thus $\lambda_{s_1} = 2.039$ $\qquad$ $\lambda_{s_2} = .518$

$E_T = 104.358$

B.4.5 Schick-Wolverton Model

The technique requires solution of the equation

$$\sum_{i=1}^{n} \frac{1}{N-i+1} = \frac{n\Sigma t_i^2}{\Sigma(N-i+1)t_i^2}$$

where all variables and parameters are previously defined with t_i replacing x_i).

Using the same data as employed above for the two de-eutrophication models the following are obtained:

$$\sum_{i=1}^{31} t_i^2 = 18(.4906)^2 + 13(.7646)^2 =$$

$$= 11.932$$

$$\Sigma(N-i+1)t_i^2 = N \sum_{i=1}^{n} t_i^2 - \sum_{i=1}^{31} (i-1)t_i^2$$

$$= 11.932N - \left(\sum_{i=1}^{17} i\right)(.4906)^2 - \left(\sum_{i=18}^{30} i\right)(.7646)^2$$

$$= 11.932N - 219.27$$

The estimated error content is the solution to

$$\sum_{i=1}^{31} \frac{1}{N-i+1} = \frac{(11.932)(31)}{11.932N-219.27}$$

By trial and error the solution is

$$N = 41.5$$

Again since 40 errors are "banked", this corresponds to an estimate of 81.5 for the total error content. This does not appear to fit the error process very well.

B.4.6 Summary of Error Estimates

The results of all models are extremely encouraging when viewed in the large. The "gestalt" which seems most important is that "nature" does indeed relate residual errors to the MTTF in an inverse way: all models are based on this assumption in one way or another and they all produce reasonable results.

B.5 QUANTITATIVE COMPARISONS OF MTTF ESTIMATES

As noted before the estimates of MTTF at the end of test time provides a "close-in" estimate. All models, except the Geometric-Poisson, which is based on several time intervals, can be compared although they formulate estimates of MTTF in different ways. The comparison can be made against the realized MTTF for the time (or times) concerned.

B.5.1 De-Eutrophication MTTF Estimate

Using the estimates for $N = 63.4$ and $\phi = .035$, the natural estimate for MTTF at the end of the two-day sample is the reciprocal of the hazard rate, $(N-n)\phi$.

For the two day sample (using cum CPU time of Table 1)

MTTF (24.01) = .884 (Actual Value .906).

For the three day sample ending at CPU time 31.26, there results

MTTF (31.26) = 1.049 (Actual Value .927)

B.5.2 Geometric De-Eutrophication MTTF Estimates

For this model the MTTF is the reciprocal of Dk^n where for the two-day sample $D = 2.998$, $k = .9735$ and

MTTF (24.01) = .767

For the three day sample, $D = 2.520$, $k = .9744$ and MTTF (31.26) = 1.091

B.5.3 Shooman Model MTTF Estimate

For the two-day sample, with using $E_T = 90.22$, $\varepsilon(\tau_2) = 71$, $C = .0406$,

MTTF (24.01) = 1.283 (Actual Value .906)

and for the three day sample

$E_T = 107.54$, $\varepsilon(\tau_2) = 79$, $C = .0302$

MTTF (31.26)= 1.161 (Actual .927)

B.5.4 Summary of MTTF Estimates

The following table provides comparison for the MTTF estimates.

Table 2

MTTF ESTIMATE COMPARISON

CPU Time	De-Eut.	Geom. De-Eut.	Shooman	Actual
24.01	.884 (2.4%)	.767 (15.3%)	1.283 (41.6%)	.906
31.25	1.049 (13.5%)	1.091 (17.7%)	1.161 (28.1%)	.927

The main entries show the estimate for the corresponding model, and in the last column, the "actual" value obtained by dividing the number of errors by CPU time for the day just beyond the test truncation time.

In parenthesis are relative errors in percent.

B.6 SENSITIVITY OF ESTIMATES

A proper comparison of the models with respect to their robustness of their estimates in the presence of changes in assumptions can best be done by simulation. However, a very simple indication of the behaviour can be found by employing properties of maximum likelihood estimates. In particular the variance of the estimates can be approximated by means of an asymptotic formula developed by R. A. Fischer. This formula and the separate analysis for each model are given in the following sections.

B.6.1 De-Eutrophication Process

The analysis for this model is based on the following likelihood function:

$$L(X_1,X_2,\ldots,X_n;N,\phi) = \prod_{i=1}^{n} \phi[N-(i-1)] \exp\{-\phi\,[N-(i-1)]X_i\} \qquad (10)$$

where ϕ and N are the parameters previously defined and X_i is the time separation between the (i-1)st and ith error.

By partial differentiation of the logarithm of the likelihood function the Maximum Likelihood Equations (MLE's) can be formed. They are:

$$\partial \log L/\partial N = \sum_{i=1}^{n} \frac{1}{N-(i-1)} - \sum_{i=1}^{n} \phi X_i = 0 \qquad (11)$$

and

$$\partial \log L/\partial \phi = \frac{n}{\phi} - \sum_{i=1}^{n} [N-(i-1)]X_i = 0 \qquad (12)$$

The variability of the estimates becomes of interest when an attempt is made to compare different models. Obviously the comparisons of models must be done on the basis of additional factors and by repeated applications on similar data. Nonetheless, other (unspecified) things being equal, the model which provides the smaller variation (standard deviation) is preferred to others.

Unfortunately, because of the implicit nature of the solution to the MLE's, the probability distribution(s) (joint, or marginal) for N and ϕ cannot be obtained, but this difficulty can to a degree be circumvented.

The general properties of maximum likelihood estimates can be used in a purely formal way to derive some measure of the variability in the estimates. This point must be emphasized since it is manifest that the use of asymptotic formulas (involving large sample sizes) on samples which are fundamentally limited to be finite (there can be no larger samples than there are errors) can result only in caution-laden approximations. Nonetheless, the experiences which have been gained using the models seem to indicate that these approximations for the variances are generally much too high.

The basis for the development of the large sample estimates is a theorem due to R. A. Fisher which states that under certain "general conditions", which have to do with the boundedness of the first three derivatives of the likelihood, the variance and covariances of the estimates are given by the inverse of a matrix formed from the mathematical expectation of second partial derivatives. Explicitly the matrix A_{ij} (which is to be inverted) in the estimation of several parameters ($_1$, $_2$,..., $_n$) has the terms

$$A_{ij} = -E\left[\frac{\partial^2 \log L}{\partial \theta_i \partial \theta_j}\right] \qquad (13)$$

where L is the likelihood function and θ_i and θ_j are two of the parameters. From Equations (11) and (12) above

$$\frac{\partial^2 L}{\partial N^2} = -\sum_{i=1}^{n} \frac{1}{(N-i+1)^2} \qquad (14)$$

$$\frac{\partial^2 L}{\partial N \partial \phi} = \frac{\partial^2 L}{\partial \phi \partial N} = - \sum_{i=1}^{n} X_i \qquad (15)$$

$$\frac{\partial^2 L}{\partial \phi^2} = - \frac{n}{\phi^2} \qquad (16)$$

And since

$$E(X_i) = \frac{1}{(N-i+1)\phi}$$

the matrix elements become:

$$A_{11} = \sum_{i=1}^{n} \frac{1}{(N-i+1)^2} \qquad (17)$$

$$A_{12} = A_{21} = \sum_{i=1}^{n} \frac{1}{(N-i+1)\phi} \qquad (18)$$

$$A_{22} = \frac{n}{\phi^2} \qquad (19)$$

where for evaluation in practical situations, the values of $\hat{N}$ and $\hat{\phi}$ (the estimates based on the data) are used.

The 2x2 variance/covariance matrix can be simply computed.

The determinant (denoted Det $_1$) of the A-matrix is

$$Det_1 = A_{11}A_{22} - A_{12}A_{21} = \sum_{i=1}^{n} \frac{1}{(N-i+1)^2} \cdot \frac{n}{\phi^2} - T^2 \qquad (20)$$

where we have used the fact that "on the average",

$\sum_{i=1}^{n} \frac{1}{(N-i+1)\phi}$ = T, the total observation time.

Hence

$$Var\ (\hat{N}) = \frac{n}{\phi^2} \cdot \frac{1}{Det_1} \qquad (21)$$

$$\text{Var}\,(\hat{\phi}) = \sum_{i=1}^{n} \left[\frac{1}{N-i+1}\right]^2 \cdot \frac{1}{\text{Det}_1} \tag{22}$$

$$\text{Covar}\,(\hat{N},\hat{\phi}) = -\frac{T}{\text{Det}_1} \tag{23}$$

Since for a fixed sample size n, the solutions for N and ϕ by means of Equations (11) and (12) depend only on the ratio $R = \frac{\Sigma(i-1)X_i}{\Sigma X_i}$, it is possible to tabulate solutions as well as the variance and covariance. This is illustrated by Table III which shows the values for a sample size n=26.

In order to tabulate the parameters for an arbitrary process it is necessary that the scale for time be normalized. Since the total observation time, T, is assumed recorded by the data collection process, it is a natural scale factor to use. It must be pointed out however, that this time is a random variable; although it is treated as if it were a constant, this is a purely pragmatic interpretation. A reasonable interpretation which can be made is that the results which are recorded are conditional on the observed time.

Given the ratio R, the MLEs become

$$\sum_{i=1}^{n} \frac{1}{N-(i-1)} = \frac{n}{N-R} \tag{24}$$

and

$$\phi T = \frac{n}{N-R} \tag{25}$$

Equation (24) can be solved essentially by trial and error. Once N is established the quantity ϕT can be obtained from Equation (25). The quality ϕT is entered in column 3 of the table.

The variance of $\hat{N}$ can be obtained in the following way:

Table III

n = 26

Ratio	Error Content	(PHI)T	DEVN	DEVφ (Normed)	COVAR (Normed)	MTTF (Normed)
14.0	51.19	.6991	35.88	.6883	-24.2005	.0568
14.2	46.94	.7942	27.74	.6907	-18.6666	.0601
14.4	43.62	.8899	22.04	.6936	-14.7968	.0638
14.6	40.95	.9866	17.87	.6966	-11.9618	.0678
14.8	38.78	1.0842	14.75	.7001	-9.8426	.0722
15.0	36.98	1.1826	12.36	.7041	-8.2155	.0770
15.2	35.47	1.2824	10.47	.7083	-6.9311	.0823
15.4	34.19	1.3836	8.95	.7129	-5.9020	.0882
15.6	33.10	1.4857	7.73	.7181	-5.0736	.0948
15.8	32.15	1.5898	6.72	.7235	-4.3850	.1022
16.0	31.34	1.6953	5.88	.7296	-3.8158	.1105
16.2	30.62	1.8027	5.17	.7361	-3.3350	.1200
16.4	30.00	1.9121	4.56	.7432	-2.9274	.1308
16.6	29.45	2.0236	4.05	.7508	-2.5795	.1433
16.8	28.96	2.1377	3.60	.7591	.2784	.1579
17.0	28.53	2.2541	3.21	.7681	-2.0185	.1750
17.2	28.15	2.3737	2.87	.7777	-1.7909	.1956
17.4	27.82	2.4959	2.58	.7883	-1.5935	.2205
17.6	27.52	2.6222	2.32	.7995	-1.4173	.2516
17.8	27.25	2.7519	2.08	.8119	-1.2630	.2912
18.0	27.01	2.8862	1.87	.8251	-1.1250	.3436
18.2	26.80	3.0247	1.69	.8396	-1.0035	.4154
18.4	26.61	3.1680	1.52	.8556	-.8964	.5200

Column 1 is the ratio $\Sigma(i-1)X_i \;/\; \Sigma X_i$

Column 2 is the estimate for the total error content

Column 3 is the normed estimate for step size: in order to determine the actual estimate of the step size, the entry in this column should be divided by the total observation time T.

Column 4 is the approximate standard deviation of the estimate of the total error content.

Column 5 is the normed standard deviation of the estimate of the step size: in order to obtain the actual standard deviation the entry in this column should be divided by the total time T.

Column 6 is the normed covariance between N and ϕ: in order to obtain the actual estimated covariance the entry should be divided by T.

Column 7 is the normed MTTF and in order to obtain the actual value the entry should be multiplied by T.

by Equation (21)

$$\mathrm{Var}(\hat{N}) = \frac{n}{\phi^2 \mathrm{Det}_1} \tag{26}$$

but using the substitution

$$S_2 \leftarrow \sum_{i=1}^{n} \frac{1}{(N-i+1)^2} \tag{27}$$

the determinant of Equation (20) can be expressed as

$$\mathrm{Det}_1 = \frac{nS_2}{\phi^2} - T^2$$

or

$$\phi^2 \mathrm{Det}_1 = nS_2 - (\phi T)^2.$$

Hence the denominator of Equation (26) can be evaluated using the estimates $\hat{\phi}T$ and $\hat{N}$.

Hence

$$\mathrm{Var}(\hat{N}) = \frac{n}{nS_2 - (\phi T)^2} \tag{28}$$

The standard deviation is the more useful measure and is obtained by taking the square root of Var $(\hat{N})$. This is entered in column 4 of the table.

The variance of $\hat{\phi}T$ is obtained in much the same way: Det_1 is evaluated, as before, and with S_2 as defined,

$$\mathrm{Var}(\hat{\phi}T) = T^2 \frac{S_2}{\mathrm{Det}_1} = \frac{S_2 T^2}{S_2\left(\frac{n}{\phi^2}\right) - T^2}$$

$$= \frac{S_2(\phi T)^2}{nS_2 - (\phi T)^2} \tag{29}$$

The standard deviation of this quantity is computed and listed in column 5 of the table. As noted a footnote the standard deviation of ϕ is obtained by dividing the column entry by T.

The covariance between N and ϕT is obtained by

$$\text{Covar}\ (N,\phi T) = -\frac{T^2}{\text{Det}_1} = -\frac{(\phi T)^2}{nS_2 - (\phi T)^2} \tag{30}$$

Again for the covariance between N and ϕ, the entry in column 6 should be divided by T.

The MTTF of the "next" error is given by $[(N-n)\phi]^{-1}$ and is estimated by employing $\hat{N}$ and $\hat{\phi}$. The value entered in the last column of the table (for n=26) is $[(\hat{N}-26)\hat{\phi}T]^{-1}$ and so must be multiplied by the user-found T.

While it is possible to formally express the variance of the estimate of the MTTF in terms of the variances and the covariance of the two estimates this is a step which will not be taken as the cascade of approximations is already too long.

In the next section where the model has estimates which have legitimate asymptotic properties, the variance of the MTTF-estimate is given.

B.6.2 Geometric De-Eutrophication Process

The analysis parallels that made for the De-Eutrophication Process. It is important to note, however, that this process has an unlimited number of errors. The likelihood function for the sample $X_1, X_2, \ldots, X_n$, is

$$L = \prod_{i=1}^{n} Dk^{i-1} \exp\{-Dk^{i-1}X_i\} \tag{31}$$

and its logarithm is

$$\log L = n\log D + \sum_{i=1}^{n} \log k^{i-1} - D \sum_{i=1}^{n} k^{i-1}X_i \tag{32}$$

The MLE's are obtained by differentiation and reduce to the two equations

$$\frac{n}{D} = \sum_{i=1}^{n} k^{i-1}X_i \tag{33}$$

and

$$\frac{1}{k}\sum_{i=1}^{n}(i-1) = D\sum_{i=1}^{n}(i-1)k^{i-2}X_i. \tag{34}$$

D can be eliminated from both equations to leave a single equation.

$$\frac{\sum_{i=1}^{n} i\, k^{i-1} X_i}{\sum k^{i-1} X_i} = \frac{n+1}{2} \tag{35}$$

Then using the solution, denoted $\hat{k}$, the estimate for D is

$$D = \frac{n}{\sum \hat{k}^{i-1} X_i} .$$

The variance and covariances for this process are found by the procedure described above. As noted, above, however, this process has an infinite number of errors, and so the sample size can become large, and the asymptotic formulas can be applied without apology.

Directly by differentiation

$$\frac{\partial^2 \log L}{\partial D^2} = -\frac{n}{D^2} \tag{36}$$

$$\frac{\partial^2 \log L}{\partial D \partial k} = \frac{\partial^2 \log L}{\partial k \partial D} = -\sum_{i=1}^{n} (i-1) k^{i-2} X_i \tag{37}$$

$$\frac{\partial^2 \log L}{\partial^2 k} = -\frac{1}{k^2} \sum_{i=1}^{n} (i-1) - D \sum_{i=1}^{n} (i-1)(i-2) k^{i-3} X_i \tag{38}$$

Since $E(X_i) = \frac{1}{Dk^{i-1}}$, the associated A-matrix elements are

$$A_{11} = \frac{n}{D^2} \tag{39}$$

$$A_{12} = A_{21} = \frac{1}{Dk} \sum_{i=1}^{n} (i-1) = \frac{1}{Dk} \frac{n(n-1)}{2} \tag{40}$$

$$A_{22} = \frac{1}{k^2} \sum_{i=1}^{n} (i-1) + \frac{1}{k^2} \sum_{i=1}^{n} (i-1)(i-2)$$

$$= \frac{1}{k^2} \sum_{i=1}^{n} (i-1)^2 = \frac{1}{6k^2} n(n-1)(2n-1) \tag{41}$$

Using Det_2 to represent the determinant of the A-matrix, we obtain after simple reduction:

$$Det_2 = \frac{1}{D^2k^2} \cdot \frac{n^2(n^2-1)}{12} \tag{42}$$

Thus the variances and covariances are

$$\text{Var } \hat{D} = D^2 \frac{2(2n-1)}{n(n+1)} \tag{43}$$

$$\text{Var } \hat{k} = k^2 \frac{12}{n(n^2-1)} \tag{44}$$

$$\text{Covar } (\hat{D},\hat{k}) = -Dk \frac{6}{n(n+1)} \tag{45}$$

In the limit these variances tend to zero. On the other hand, it will be noted that the correlation coefficient between the estimates is quite high (in absolute value):

$$\rho = -\sqrt{\frac{3}{2} \frac{(n-1)}{(2n-1)}} \tag{46}$$

which is in excess of 0.85.

The estimate for the MTTF which has the character of the maximum likelihood estimates is given by

$$M_2 = \frac{1}{\hat{D}\hat{k}^n}$$

where the subscript 2 denotes the estimate for the GDEM. The asymptotic approximations can be employed in another reasonable approximation in order to derive a measure of the variation in the estimate of the MTTF. By differentiation taking the total differential and expectations it is seen that (47)

$$\text{Var } M_2 = \frac{1}{D^2k^{2n}} \text{Var}(D) + \frac{2n}{D^3k^{2n+1}} \text{Covar } (D,k) + \frac{n^2}{D^2k^{2n+2}} \text{Var}(k)$$

where, again the estimates would be used as proxies for the (unknown) parameters.

B.6.3 Geometric-Poisson Model

From Equation 10 of Reference 2, the likelihood function is

$$L(n_1,n_2,\ldots,n_m) = \prod_{i=1}^{m} (\lambda k^{i-1})^{n_i} \exp(-\lambda k^{i-1}) \cdot \frac{1}{n_i!} \tag{48}$$

and

$$\log L = \sum_{i=1}^{m} n_i(\log\lambda+(i-1)\log k) - \sum_{i=1}^{n} \lambda k^{i-1} - \sum_{i=1}^{n} \log n_i! \tag{49}$$

Hence

$$\frac{\partial^2 \log L}{\partial \lambda 2} = - \frac{1}{\lambda^2} \sum_{i=1}^{m} n_i \tag{50}$$

$$\frac{\partial^2 \log L}{\partial k \partial \lambda} = \frac{^2\log L}{\partial \lambda \partial k} = - \sum_{i=1}^{m} (i-1)k^{i-2} \tag{51}$$

$$\frac{\partial^2 \log L}{\partial_k{}^2} = \frac{-1}{k^2} \sum_{i=1}^{m} (i-1)n_i - \lambda \sum_{i=1}^{m} (i-1)(i-2)k^{i-3} \tag{52}$$

The variables n_i are assumed by this model to be Poisson distributed, and so

$$E(n_i) = \lambda k^{i-1} \tag{53}$$

Accordingly

$$A_{11} = \frac{1}{\lambda} \sum_{i=1}^{m} k^{i-1} \tag{54}$$

$$A_{21} = A_{12} = \sum_{i=1}^{n} (i-1)k^{i-2} \tag{55}$$

$$A_{22} = \lambda \Sigma (i-1)^2 k^{i-3} \tag{56}$$

The variances and covariances are therefore

$$\text{Var}\,\hat{\lambda} = \frac{\lambda}{\text{Det}_3} \cdot \sum_{i=1}^{m} (i-1)^2 k^{i-3} \tag{57}$$

$$\text{Var}\ \hat{k} = \frac{\lambda^{-1}}{\text{Det}_3} \cdot \sum_{i=1}^{m} k^{i-1} \tag{58}$$

$$\text{Covar}\ (\hat{\lambda},\hat{k}) = \frac{1}{\text{Det}_3} \sum_{i=1}^{m} (i-1)k^{i-2} \tag{59}$$

B.6.4 Shooman Model

M. L. Shooman in Reference 5, gives the following expressions for the maximum likelihood equations for the two parameters or E_T and C (originally K):

$$\hat{C} = \frac{n_1 + n_2}{\left[\frac{E_T}{I_T} - E_C(\tau_1)\right] H_1 + \left[\frac{E_T}{I_T} - E_C(\tau_2)\right] H_2} \quad \text{[Eq. 21 of Ref. 5]} \tag{60}$$

and

$$\hat{C} = \frac{1}{H_1+H_2} \left[\frac{n_1}{\left[\frac{E_T}{I_T} - E_C(\tau_1)\right]} + \frac{n_2}{\left[\frac{E_T}{I_T} - E_C(\tau_2)\right]}\right] \quad \text{[Eq. 22 of Ref. 5]} \tag{61}$$

where n_1 and n_2 are "the number of runs used in testing for times H_1 and H_2 respectively". (This is incorrect; they should be the number of unsuccessful runs as will be shown herein).

Shooman refers to another of his papers for the justification and development, unfortunately there is no discussion in the referenced paper with respect to maximum likelihood estimation [Reference 5]. In order to expedite the discussion and further clarify the Shooman Model, the following analysis is provided.

Fundamental to the analysis is the assumption that the software error genera ion is governed by a Poisson point process whose intensity (failure rate) is proportional to the current number of errors. For short periods of time, (H_{s1} and H_{s2} in the present instance) the intensity factor can be assumed to be proportional to the number of remnant errors, which in Shooman's notation is

$$\lambda_1 = C\left(\frac{E_T}{I_T} - \varepsilon_C(\tau_1)\right.$$

for the first period, and

$$\lambda_2 = C\left(\frac{E_T}{I_T} - E_C(\tau_2)\right)$$

The number of unsuccessful runs occurring in a time H_s is then given by the Poisson distribution. Explicitly, the number of errors (or unsuccessful runs or whatever rare event is being counted), n_1, observed during H_{S_1}, is given by the Poisson law

$$P(N=n_1) = \frac{(\lambda_1 H_{S_1})^{n_1} e^{-\lambda_1 H_{S_1}}}{n_1!} \tag{62}$$

Thus for two runs of duration H_{S_1} and H_{S_2} with observed numbers of unsuccessful runs n_1 and n_2, the likelihood function is

$$L = \frac{(\lambda_1 H_{S_1})^{n_1} e^{\lambda_1 H_{S_1}}}{n_1!} \cdot \frac{(\lambda_2 H_{S_2})^{n_2} e^{-\lambda_2 H_{S_2}}}{n_2!} \tag{63}$$

The maximum likelihood equations obtained by differentiating the log-likelihood are

$$\frac{\partial \log L}{\partial C} = \frac{n_1}{C} + \frac{n_2}{C} - H_1 R_1 - H_2 R_2 = 0 \tag{64}$$

and

$$\frac{\partial \log L}{\partial E_T} = \frac{n_1}{I_T R_1} + \frac{n_2}{I_T R_2} - \frac{1}{I_T} H_{S_1} C - \frac{1}{I_T} H_{S_2} C = 0 \tag{65}$$

where $R_i = \left[\frac{E_T}{I_T} - E_C(\tau_1)\right]$ is used for convenience, and H_{S_1} is now replaced by H_1

Solving for C in each equation produces:

$$\hat{C} = \frac{n_1+n_2}{H_1 \hat{R}_1 + H_2 \hat{R}_2} \tag{66}$$

and

$$\hat{C} = \frac{1}{H_1+H_2} \left[\frac{n_1}{\hat{R}_1} + \frac{n_2}{\hat{R}_2} \right] \tag{67}$$

and where $\hat{R}_i$ and $\hat{C}$ are the estimators of C and R_i respectively. These are the equations Shooman reports.

Following the same steps employed in the preceding analyses with respect to development of variances and covariances, the second partials are taken.

$$\frac{\partial^2 \log L}{\partial C^2} = -(n_1+n_2) \frac{1}{C^2} \tag{68}$$

$$\frac{\partial^2 \log L}{\partial C \partial E_T} = \frac{-1}{I_T} (H_1+H_2) = \frac{\partial^2 \log L}{\partial E_T \partial C} \tag{69}$$

$$\frac{\partial^2 \log L}{\partial E_T^2} = - \frac{n_1}{I_T^2 R_1^2} - \frac{n_2}{I_T^2 R_2^2} \tag{70}$$

The expectation of n_i is

$\lambda_i H_i = H_i C R_i$, so that the A matrix elements are:

$$A_{11} = \frac{1}{C} [H_1 R_1 + H_2 R_2] \tag{71}$$

$$A_{12} = A_{21} = \frac{1}{I_T} (H_1+H_2) \tag{72}$$

$$A_{22} = \frac{C}{I_T} 2 \left[\frac{H_1}{R_1} + \frac{H_2}{R_2} \right] \tag{73}$$

Hence, from general relations,

$$\text{Var } \hat{E}_T = \frac{1}{\text{Det}_4} \cdot \frac{1}{C} \left[H_1 \cdot R_1 + H_2 \cdot R_2\right] \tag{74}$$

$$\text{Var } \hat{C} = \frac{1}{\text{Det}_4} \left(\frac{C}{I_T}\right)^2 \left[\frac{H_1}{R_1} + \frac{H_2}{R_2}\right] \tag{75}$$

$$\text{Covar } (\hat{C}, \hat{E}_T) = -\frac{1}{\text{Det}_4} \cdot \frac{1}{I_T} (H_1 + H_2) \tag{76}$$

and

$$\text{Det}_4 = \frac{1}{I_T^2} \left[(H_1 \cdot R_1 + H_2 \cdot R_2) \cdot \left[\frac{H_1}{R_1} + \frac{H_2}{R_2}\right]\right] - \frac{1}{I_T^2} (H_1 + H_2)^2 \tag{77}$$

In Reference 5, Shooman gives the asymptotic relations

$$\text{Var } \hat{C} \underset{n}{\to} \frac{C^2}{n_1 + n_2} \qquad \text{(Eq. 23 of Ref. 5)}$$

and (in our notation)

$$\text{Var } \hat{E}_T \underset{n}{\to} \frac{I_T^2 R_1^2 R_1^2}{n_1 R_1^2 + n_2 R_2^2} \qquad \text{(Eq. 24 of Ref. 5)}$$

These do not appear to be the same or similar to the corresponding expressions (74) and (75). If it is assumed that the second term in expression (76) is much smaller than the first term, and can be ignored, then by substitution

$$\text{Var } \hat{C} \cong \frac{C}{H_1 R_1 + H_2 R_2} \tag{78}$$

If the expected values for n_1 and n_2 are substituted into the Shooman expression these are seen to be the same. There is no need to have n large, as Shooman implies in his formulation, but it is necessary for the second term to be small.

Correspondingly, using the same approximation for Det_4

$$\text{Var } \hat{E}_T \cong \frac{I_T^2}{C} \left[\frac{H_1}{R_1} + \frac{H_2}{R_2}\right]^{-1} \tag{79}$$

If C is replaced by its estimator given in Equation (66) above, this becomes

$$\text{Var } \hat{E}_T \simeq I_T^2 C \frac{R_1^2 R_2^2}{n_1 R_2^2 + n_2 R_1^2} \tag{80}$$

This is somewhat similar to the Shooman expression but differs in a very interesting way: the terms involving products of n and R are "mixed", i.e., one factor is at τ_1, and the other at τ_2.

There is an independent way of verifying expression (77). From the knowledge of the Poisson-law, it is true that the variance of N_1 is equal to the parameter $\lambda_1 H_{S_1}$, so that equation (65) can be used to obtain

$$\text{Var } C = \frac{1}{[H_1 R_1 + H_2 R_2]^2} x[\text{Var}(n_1) + \text{Var}(n_2)]$$

(there is independence between n_1 and n_2) and so

$$\text{Var } C = \frac{C}{H_1 R_1 + H_2 R_2}$$

which is equation (77).

There is no explicit solution for E_T so that a similar validation does not seem possible.

REFERENCES

1. Z. Jelinski, P. B. Moranda. Software Reliability Research. pp. 464ff, Statistical Computer Performance Evaluation, edited by Walter Freiberger, Academic Press, 1972.

2. P. Moranda. Predictio. ; of Software Reliability During Debugging. 1975 Proceedings of the Annual Reliability and Maintainability Symposium, January 1975, Washington D.C.

3. M. L. Shooman. Probabilistic Models for Software Reliability Predictions. pp. 485.502, Statistical Computer Performance Evaluation, edited by Walter Freiberger, Academic Press 1972.

4. J. C. Dickson, J. L. Hesse, A. C. Kuentz, M. Shooman. Quantitative Analysis of Software Reliability. Proceedings of Annual Reliability and Maintainability Symposium, San Francisco, 1972.

5. M. Shooman. Operational Testing and Software Reliability During Program Development. Record of IEEE Symposium Computer Software Reliability, New York City, 1973.

6. M. Shooman. Software Reliability: Measurements and Models. Proceedings of Annual Reliability and Maintainability Symposium, Washington, D.C., 1975.

7. G. J. Schick and R. W. Wolverton. Assessment of Software Reliability. 11th Annual Meeting of German Operations Research Society, Hamburg, 1972.

8. W. L. Wagoner. The Final Report on Software Reliability Measurement Study. Aerospace Report No. TOR-0074 (4112)-1, August 1973.

9. F. Akiyama. An Example of Software System Debugging. Proceedings of IFIP Congress 1971, North Holland Publishing Company, 1972.

10. T. A. Thayer, et.al.,Software Reliability Study. TRW Interim Technical Report on Contract F30602-74-C-0036 with Rome Air Development Center, 25 June 1974.

Appendix C

PROGRAM TESTING

C.1 INTRODUCTION

For the purposes of this study, program testing will be defined as the process of verifying that a selected self-contained unit of code (such as a subroutine) complies with the requirements against which it was designed. These requirements define the functions the code should perform, the structure of the code within the unit and the environment in which the code must operate. Program testing as we have defined it does not include the testing of relationships between units of code. It is based on complete knowledge of the internal structure of the unit, as opposed to the black box approach.

It is well-known that complete testing is not feasible even in the few cases where it is possible. Therefore, the next best approach is to test as completely as possible within the constraints of economics and value received.

Various tools and techniques have been devised to aid in program verification. While none as yet can assure that the code performs correctly under all conditions to which it may be subjected, each adds to the probability that the code will perform its intended functions correctly at the time that it is called upon to do so.

A number of tools have been built that support the testing function and research continues in the development of technologies that will result in new tools and in improvements of those already in existence. Some manual techniques have been advanced which offer promise as verification aids.

The tools and techniques with which this task is concerned are those that actually interact with the code in some way as opposed to those that support testing in a peripheral fashion or those that support system testing from a functional standpoint.

This report deals with the application of these tools and techniques to the testing of programs, the research being performed to improve the technology, and an evaluation of the practicality of each type of tool or technique in today's software development environment.

C.2 MODULARIZATION

The testing of programs using currently available tools and techniques generally requires that the total system be broken down into manageable units, each of which can be considered a separate test object. Top-down design ultimately results in a set of routines for each level of decomposition.

Each routine (unit) is a test object to which the tools and techniques can be applied. Intellectual manageability of the unit is one criteria for establishing the size of the unit of code, particularly for manual techniques such as walk throughs and program proofs. A second criteria is the amount of code that can be handled by some tools such as test case generators. Other criteria may be considered such as requirements for single entry/single exit units of code. In most instances, limiting the size of the unit of code to 100[1] statements or less provides a test object that is intellectually manageable and can be handled by the test tools which are to be applied.

While modularization is a design function, its importance in program testing is so great that if not provided for in the design phase, it must be considered in the testing phase. Decomposition of programs, while generally based on structure, must also take logical processes into consideration.

C.3 MANUAL TECHNIQUES

Walk-throughs

The careful reading of a program by someone other than the programmer with the objective of evaluating its correctness with respect to a given specification has proved to be effective, as documented by Weinberg[1] and Baker[2].

The person reading the program can detect errors transparent to the programmer, because he is not psychologically biased by his identification with the program. The programmer who made the error will often consistently overlook it because he is reading the program from the same point of view as when he wrote it.

The process of cooperative program checking is called egoless programming by Weinberg because it eliminates the ego identification the programmer has with the object of his creation and allows an impartial evaluation.

The effectiveness of a walk-through is also a function of the ease with which the program can be understood. Since reading the program for correctness is in effect a mental proof of correctness, it is imperative that the unit of code be small enough to be understood. The mental execution of the program differs from program proving in the degree of formality.

Walk-throughs can be performed at various stages of coding depending on the complexity of algorithm being implemented. In very complex code, such as that which must meet extremely tight efficiency requirements, it may be advantageous to walk through the detailed design, the initial implementation, and the final refinement in order to assure understandability and reduce the chance of implicit or hidden errors.

Program proving

Program proving, both formal and informal, is discussed in Appendix E.

When program proving was first seriously advanced as a candidate for automation, it was thought that the difficulties inherent in this approach to program verification could be overcome given the time to consider them properly. As late as 1972 in a report by Information Research Associates,[3] it was stated that "it seems possible that within the next two to five year period to bring this to a fairly respectable state of automatic analysis". However, to date, the difficulties have not been found to be surmountable, and it appears that automatic program proving still faces some formidable problems demanding further research.

As a manual technique, program proving can be considered from two points of view. The first is from the point of view of proving an algorithm correct and can be performed during the design stage. The second is from the point of view of verifying the design representation in code and is performed during the coding stage.

For units of code containing more than a few branches and/or loops, the manageability of the proof becomes virtually impossible. Therefore, while the theory behind program proving is viable, its use is not feasible until ways are devised to more fully automate the process in a workable fashion.

C.4 AUTOMATED TECHNIQUES

A variety of automated tools to support program testing have been built. In addition, a concerted effort is in progress to build tools with new capabilities or to add improved capabilities to the tools now in existence. A comprehensive evaluation of the software requires both static analysis to evaluate structural characteristics, and dynamic analysis to evaluate behavioral characteristics. While neither type verifies that the unit of code performs the functions specified for it, both provide useful information about the testability of the code.

An interesting concept should be mentioned here, however. The kernel of functional requirements verification is the dynamic analysis of code if it can be determined that the unit meets the requirements defined by assertions added to the code. This capability is proposed as an enhancement to MDAC's PET[4] program, and is discussed in greater detail later in this report. An evaluation of the capabilities of various tools now in existence was made in the performance of this study and is contained in Appendix A. This task will consider the technology that supports these tools as well as others being developed in research projects.

Standards Checkers

The test object of standards checkers is the set of source statements in a self-contained unit of code such as a subroutine or a procedure. They check the statement set for conformance to predefined standards such as adequate and consistent commentary, statement positioning relative to the entire set (e.g. placing declaratives in a specified order, placing format statements together before or after the executable code, or placing internal procedures before executable code (in a procedure-oriented language), and program length.

TRW's Code Auditor[5] program currently checks for 38 programming constraints in FORTRAN code.

Applied Data Research (ADR)[6] has COBOL standards checking capability in their Metacobol system.

Computer Software Analysts, Inc. (CSA)[7] markets a product called Standards Auditor which is available for both FORTRAN and COBOL source programs.

Bell's PFORT Verifier[8] checks FORTRAN code for conformity to ANSI standards.

While it may be argued that these tools are not directly involved in program testing, they impose an orderliness upon the program that is designed to eliminate errors caused by haphazard program construction. This orderliness also contributes directly toward the program's understandability and maintainability.

Automatic Test Case Generation

The overview testing of software has been called an art because the selection of a set of test cases, that will adequately test a program with carefully chosen input data to minimize the number of cases and maximize the value of their application, requires a great deal of insight and cleverness. The development of a methodology to automate this process removes it from the realm of art and the implicit errors and omissions that are inherent.

Several systems are presently being developed to automatically generate test cases. L. Stucki, MDAC and W. Howden[9,10,11,12] of the University of California at San Diego are developing a test case generator for MDAC under contract to the National Bureau of Standards. R. Hoffman[13,14,15,16] of TRW in Houston is developing the Automated Test Data Generator for the NASA/Johnson Space Center in Houston. E. Miller[17], formerly of General Research Corporation, designed a test data generator for commercial use in their automated tool collection called RXVP. L. Clarke[18] of the University of Colorado is developing an automatic test case generator supported in part by an NSF grant. All of the above support FORTRAN programs, and are designed to generate test data based on an examination of the syntax. Other types of test data generators are in use, and are basically driven by input parameters supplied by the programmer, with no direct knowledge of the source code being required. This report will address the work by Hoffman, Howden, Clarke and Miller since that work seems most applicable to NASA needs.

The systems listed above are presently being developed to automatically generate test data based on an analysis of the code. This syntactic analysis provides cases that depend upon the control structure of the program. The code is analyzed and the control structure identified based on the presence of predicates. These predicates (logical decisions) cause transfers to various parts of the program, both forward and backward, and create the existence of a number of alternate paths through the code.

If the branches dictated by the predicates were all independent of each other, an enormous number of paths would be possible in a program. Looping, in particular, has the greatest effect.

However, in reality, the number of paths is much smaller, particularly in a non-iterative system, because the branches tend to be dependent. That is, a branch taken on a true condition may eliminate not only the false branch but an entire section of a path including other branches that can only be reached if the false condition existed for the first branch.

The goal of the test data generators developed to date is to exercise not only all statements in a program but also all branches (It is possible to exercise all statements without having exercised all branches).

The systems to be discussed in this report all begin by defining the control structure of the code, then they eliminate paths which cannot be executed, leaving only the paths which can be traversed using some input data to drive execution down those paths.

Semantic analysis is not addressed. It is up to the programmer to determine if the cases generated to exercise the code do in fact demonstrate that the code functions as it was intended.

The usefulness of automatic test case generators lies in their ability to show that code as written can be reached when driven by some data within the input domain. The extrapolation of this information to program correctness within acceptable bounds is left to the programmer. The test case data can be analyzed to determine relevance to the areas of interest, providing the base for the proper set of test cases required to adequately exercise the program for the purposes intended.

General Approach

The McDonnell Douglas approach originally taken by Stucki and Howden[9,10,11,12] was to decompose a FORTRAN program into a finite number of standard classes of program paths, then to try to generate a set of test cases that causes one path from each class to be tested.

The methodology was divided into five phases.

1) Analysis of a program and construction of descriptions of the standard classes of paths.

2) Construction of descriptions of the sets of input data that cause the different standard classes of paths to be followed. These are implicit descriptions.

3) Transformation of the implicit descriptions into equivalent explicit descriptions.

4) Construction of explicit descriptions of subsets of the input data sets for which the third phase was unable to construct explicit descriptions.

5) Generation of input values that satisfy explicit descriptions by the application of inequality solution techniques.

In testing a program it is necessary to choose a finite set of paths that could be tested from the potentially infinite number of paths possible through the program. A boundary-interior method was used for choosing the paths. This method groups the paths through the program into a set of classes. One path in each class is tested, by which it is assumed that if a test is successful, all other paths in that class are considered correct.

The philosophy underlying the boundary-interior method is based on the assumption that a "complete set of tests must test alternative paths through the top level of a program, alternative paths through loops and alternative boundary tests of loops". A boundary test of a loop is a test which causes the loop to be entered but not iterated. An interior test causes a loop to be entered and then iterated at least once.

The boundary-interior method separates paths into separate classes if they differ other than in traversals of loops. If two paths P_1 and P_2 are the same except in traversals of loops they are placed in separate classes if

(i) one is a boundary and the other an interior test of a loop

(ii) they enter or leave a loop along different loop entrance or loop exit branches.

(iii) they are boundary tests of a loop and follow different paths through the loop.

(iv) they are interior tests of a loop and follow different paths through the loop on their first iteration on the loop.

Class descriptions consist of branch predicates, assignment statements, I/O statements and FOR-loops (to represent an arbitrary number of traversals of a loop.) The complete set of class descriptions for a program can be represented in the form of a description tree , in which the left-most path describes the class of all paths which test the interior of the loop in the program, and the other paths are boundary tests.

The description tree is formed as the program is read with alternative paths being consutrcted for each branch and each loop.

Once the set of paths has been described, the methodology of phase two constructs implicit input data descriptions of the sets of data that cause classes of paths to be followed. The predicates and predicate affecting statements are extracted from class descriptions. Output statements are deleted and all input statements are replaced by assignment statements in which dummy variables represent the values in the input stream (and the program is assumed to read the next value in the stream). Phase two also deletes all assignment statements that do not affect predicates. This is done by reading backward from each predicate and constructing lists of those variables that affect the predicates.

Phase three attempts to transform implicit input data descriptions into explicit descriptions. The assignment statements are evaluated by substituting the current symbolic values into the statement, which gives the dependent variable a current symbolic value. These values are then substituted when the variables are encountered in predicates and relations. Any of these statements that do not affect the predicates are deleted. The FOR-loops are processed in basically the same way. If a loop is closed and does not change any variable affecting predicates it can be deleted.

Problems arise when array reference and FOR-loop indexes can only be determined at execution time; when values of variables occurring inside a loop are computed outside of the loop; when values of variables occurring outside a loop are computed inside the loop; and when disjunctive and recurrence statements occur. In each of these cases the values of the assigned variables must be classified as indeterminate and cannot be deleted.

The evaluation is an iterative process, since statements in a loop may not be evaluatable until later processing. Once evaluated, re-evaluation of prior statements may be possible.

Phase four completes the transformation of implicit descriptions to explicit descriptions by choosing particular values of loop bounds and particular terms in disjunctive statements. These values are a subset of a set of values that satisfy the description.

At this point, feasibility of a description becomes an issue. A description is feasible if there are values in the input domain that satisfy the descriptions. Infeasible descriptions describe the empty subset of the input domain. Assuming the partially explicit description is feasible, Phase four attempts to choose loop bounds and disjunctive terms that result in a feasible subset description.

Phase five generates the test data. It divides the standard classes of paths into three sets: those for which it can generate test data, those for which it can determine infeasibility and those for which it can do neither.

Phase five is an integrated collection of inequality solution techniques and is based on a backtrack search. This method can be applied to both linear (Kuhn 1956) and non-linear (Howden 1972) systems.

The backtrack search starts with the last inequality and progresses up through the path, solving each inequality in sequence. If at any point, a solution cannot be found that satisfies the constraints on the variable

involved, the method backs up and attempts to change the solution to the previous inequality. If it can, then the solution process proceeds. If not, it backtracks further in the attempt to find a sequence of solutions. If it backs all the way, the system is unsolvable. If all the inequalities in the path are solved, the system is solved.

This general methodology produces a large number of test cases because of the boundary-interior approach to handling loops. Hence more recent research has been geared toward the selection of path segments of interest in order to restrict the number of test cases to be generated. This allows the examination of loops, specified sets of branches and/or loops without bearing the overhead of generating data for paths of peripheral interest. A command language has been introduced to enable the user to select the classes or subsets of classes of paths. A feature was added to include the identification of statements, loops and branches by number, allowing the user to communicate more easily with the system by naming paths and classes of paths through a program. Other features include user assertion capabilities for stating desired logical conditions and for assisting in the symbolic evaluation process. The DISSECT system which is now being experimented with represents the latest work in this area.

McDonnell Douglas

Dissect System

The DISSECT system can be used in different ways and can take on different forms, depending on the capabilities which have been implemented. In using the system the programmer begins by preparing a number of cases. Each case describes an analysis to be carried out by the system. In its simplest form, a case consists of:

(a) A procedure consisting of a sequence of commands which cause the selection from the program of a path, partial path or set of paths or partial paths. A path is defined to be any possible flow of control through the program.

(b) A set of commands which directs the system to print out:

(i) a system of predicates which describes the data that causes the selected paths to be followed; and
(ii) symbolic expressions which describe the cumulative effects of the computations carried out by the selected paths.

Several of the possible more complex features have also been implemented. The more complex features can be used to carry out more sophisticated program analyses. The current system will allow a user to conditionally carry out path selection commands based on whether the selections will cause the generation of inconsistent systems of predicates, to generate and insert assertions into the selected paths, and to manually force certain simplifications of the predicates and output for selected paths. An extended system would allow the user to combine systems of predicates, symbolic output and assertions in verification conditions; to solve

systems of predicates to generate test data; and to prove correctness by proving verification conditions. Attempts to automate those last two features have not been completely successful in specially designed test data generation and proof of correctness systems and they are not planned for inclusion in DISSECT.

In the following example the system is used to "read" a program. DISSECT can be used to help a user read a program by analyzing the program to see what computations are carried out by the program along selected paths and what data causes the paths to be selected.

Each case which is preapred for input to the DISSECT system can contain a section of text which describes in English the subdomain of the program corresponding to the case and the actions carried out by the program over the subdomain for the case. The DISSECT system can be used to confirm that the selected paths corresponding to the case are applied in the situations and carry out the actions specified in the text for the case. In helping the user to read a program, the system helps the user to confirm that the program implements its specifications.

Example

The program in Figure C-1 is called PDIV. The program comes from the IBM Scientific Subroutine Package and can be used to divide one polynomial by another. The input to PDIV consists of two vectors of polynomial coefficients of length IDIMX and IDIMY where IDIMX-1 and IDIMY-1 are the degrees of the polynomials.

It is not immediately obvious from a casual reading of this program that it carries out division of polynomials. Most of the difficulty is due to the language the program is written in and the style of programming. The program has been clearly designed to handle these classes of input:

(i) Zero divisor: IDIMY = 0;
(ii) Divisor of higher degree than dividend: IDIMY >IDIMX; and
(iii) Divisor not of higher degree than dividend: IDIMY $\leq$IDIMX.

The actions taken by the program for the first two classes of input are obvious. There is not need to construct DISSECT cases to analyze the actions and program subdomains for these classes. A user might still construct these cases in order to maintain, using DISSECT, a document containing the record of a complete examination of all of the important cases covered by PDIV. If so, he might construct the cases contained in Figure C-2 and Figure C-3. The case descriptions contained in Figures C-2 and C-3 include the output from the system. The user supplies the text description of the case and the DISSECT commands which specify a set of paths and the output to be generated for the paths. The system generates the systems of predicates and the symbolic output for the paths. If a case involves the specification of more than one path or partial path, the output for each path is grouped together under a SUBCASE heading. The output for CASE 1 in Figure C-2 involves two subcases.

```
      SUBROUTINE PDIV(P,IDIMP,X,IDIMX,Y,                          1
     1IDIMY,TOL,IER)                                              2
        DIMENSION P(1), X(1),Y(1)                                 3
        CALL PNORM (Y,IDIMY,TOL)                                  4
        IF(IDIMY)  50,50,10                                       5
   10   IDIMP=IDIMX-IDIMY+1                                       6
        IF(IDIMP) 20,30,60                                        7
C     DIVISOR DEGREE TOO LARGE                                    8
   20   IDIMP=0                                                   9
   30   IER=0                                                    10
   40   RETURN                                                   11
C       ZERO DIVISOR                                            12
   50   IER=1                                                    13
        GO TO 40                                                 14
   60   IDIMX=IDIMY-1                                            15
        I=IDIMP                                                  16
   70   II=I+IDIMX                                               17
        P(I)=X(II)/Y(IDIMY)                                      18
C       SUBTRACT MULTIPLE OF DIVISOR                            19
        DO 80 K=1,IDIMX                                          20
        J=K-1+I                                                  21
        X(J)=X(J)-P(I)*Y(K)                                      22
   80   CONTINUE                                                 23
        I=I-1                                                    24
        IF(I) 90,80,70                                           25
C       NORMALIZE REMAINDER POLYNOMIAL                          26
   90   CALL PNORM(X,IDIMX,TOL)                                  27
        GO TO 30                                                 28
        END                                                      29
```

Figure C-1. PDIV Program

```
CASE _:  DIVISOR TOO LARGE
 OUTPUT:  PATH, PREDICATES.
 PATHS:     DEFAULT SELECT ALL CONSISTENT;
          5  SELECT  .GT.;
          7  SELECT  .LE.;

SUBCASE 1.1:
    PATH:    1-7, 9-11
    PREDICATES:      5  IDIMY .GT.    0
                     7  IDIMY-IDIMY+1  .LT.  0.
                    11  RETURN

SUBCASE 1.2:
   PATH:     1-7, 10, 11
   PREDICATES:        5  IDIMY .GT.  0
                      7  IDIMY-IDIMY+1  .EQ.  0
                     11  RETURN
```

Figure C-2. CASE 1 -- DIVISOR too large

```
CASE 2   ZERO DIVISOR
 OUTPUT:  PATH, PREDICATES.
 PATHS:      DEFAULT SELECT ALL CONSISTENT;
          5  SELECT  .EQ.;
       PATH:  1-5, 13, 14, 11.
       PREDICATES:      5  IDIMY  .EQ.  0
                       11 RETURN
```

Figure C-3. CASE 2 -- Zero divisor

Programs to be analyzed by DISSECT must have sequence numbers or line numbers. The text in the CASE section of CASE 1 describes the subset of the input domain consisting of pairs of polynomials where the divisor is of larger degree than the dividend. The path selection commands in the PATHS section describe the set of paths which the user claims to correspond to this case. The 5 SELECT .GT. command causes the .GT. branch to be chosen in statement 5. The DEFAULT SELECT ALL command tells the system what to do if it encounters a conditional branching statement for which there is no associated SELECT command. The command instructs the system to follow all branches which do not cause the system of predicates associated with the path that follows a branch to become inconsistent. The system analyzes the program and produces the output in the SUBCASE sections. In more complicated cases the output will be useful in determining whether the program conforms to the program specifications in the CASE text descriptions.

The action taken by the program for pairs of polynomials where the divisor is of degree less than or equal to the degree of the dividend is more obscure. The user may wish to confirm that the program works correctly for this class of data by constructing cases for which: the divisor and the dividend both have minimal degree -- degree one; they both have the same non-minimal degree, say degree 2; and the divisor has smaller degree than the dividend, say divisor degree 2 and dividend degree 3. Figure C-4 contains the CASE corresponding to the output for which the divisor and the divident both have degree 2. Figure C-5 contains the CASE corresponding to the input for which the divisor has degree 2 and the dividend degree 3. In both cases the user has requested that the system print out the path sequence numbers and the symbolic output for the paths. The ASSIGN command allows the user to give values to variables in paths. The result of the ASSIGN commands in CASE's 3 and 4 is to cause the variable IDIMX and IDIMY to be replaced with constants in the symbolic output for the CASES.

The LOOP commands in CASES 3 and 4 specify how many times a loop in a path should be executed. The command 25 SELECT .GT., .LE.; specifies that the .GT. branch in statement 25 should be followed during the first iteration of the loop containing the statement and the .LE. branch the second time.

The system also allows explicit conditional commands. The conditions in these commands can involve "loop counts", values of program variables, and path attributes. Path attributes are character strings which can be attached to any program statement. All paths which pass through a statement having an attribute acquire that attribute.

The DISSECT system allows a user to isolate parts of his program and to apply automatic program analysis tools that will help him to see what circumstances cause those parts of the program to be executed. He can also apply analysis tools to see what computations are carried out by different parts of the program and to compare those with program specifications. More complicated analysis tools can be incorporated into DISSECT to allow a user to ask for the automatic generation of test data, the formation of verification conditions and the proof of program properties.

```
CASE 3:  X AND Y HAVE SAME DEGREE - 2
OUTPUT:    PATH, OUTPUT (P,X)
PATHS:     ASSIGN IDIMX = 3;
           ASSIGN IDIMY = 3;
         5    SELECT  .GT.;
         7    SELECT  .GT.;
        20    LOOP  2;
        25    SELECT  .LE.;

     PATH:  1-7, 15-23, 20-23, 20, 24-28, 10, 11.
     OUTPUT:
     ARRAY P:
       P(1) = X(3)/Y(3)
     ARRAY X:
       X(1) = X(1) - (X(3)/Y(3)) * Y(1)
       X(2) = X(2) - (X(3)/Y(3)) * Y(2)
```

Figure C-4. CASE 4 -- Equal Degrees

```
CASE 4:     X HAS GREATER DEGREE THAN Y - 3 AND 2.
OUTPUT:   PATH, OUTPUT(P,X)
PATHS:      ASSIGN,IDIMX = 4;
            ASSIGN IDIMY = 3;
       5    SELECT  .GT.;
       7    SELECT  .GT.;
      20    LOOP  2;
      25    SELECT  .GT., .LE.;

   PATH:  1-7, 15-23, 20-23, 20, 24, 25, 17-23,
          20-23, 20, 24-28, 10, 11.
   OUTPUT:
    ARRAY P:
      P(1) = (X(3) - (X(4)/Y(3)) * Y(2))/Y(3)
      P(2) = X(4)/Y(3)
    ARRAY X:
      X(1) = X(1) - ((X(3) - (X(4)/Y(3))
               * Y(2))/Y(3)) * Y(1)
      X(2) = X(2) - (X(4)/Y(3)) * Y(1) - ((X(3)
               - (X(4)/Y(3)) * Y(2))/Y(3)) * Y(2)
      X(3) = X(3) - (X(4)/Y(3)) * Y(2)
```

Figure C-5. CASE 5 -- Degree of divisor less than degree of dividend

General Research

Miller[17] has also developed a system that generates test cases for FORTRAN programs. The first step is the decomposition (if necessary) of the program into a series of smaller segments that can be dealt with separately. This process decomposes the program structure into segments with the property that each has a single entry single exit.

Miller's approach is to construct a tree representation of the program by automatically performing a series of reductions of the program directed graph. This corresponds pretty much to Howden's reduction of the program to a state diagram of the class descriptions.

Once the tree representation has been constructed, the structure (program control) is analyzed to determine the program execution patterns (paths).

This tool also uses the backtracking method to construct test cases. However, a basic difference between it and Howden's method is the handling of loops and the extent of processing performed during backtracking. While Howden derives his entire set of inequalities prior to the backtracking process, Miller performs much of the derivation during backtracking. Howden attempts to handle loops as part of the system to be solved, while Miller reduces loops to non-iterative flow.

The backtracking in Miller's system implements a set of rules that dictate the actions taken by the backtracker as it traverses the statement sequence in reverse order. The reduction of the program produces a set of simultaneous non-linear inequalities involving only input-space variables and constants. These inequalities are now solved and the result is the test case.

In the case of iterative flow, manual intervention may be necessary if a valid test case set cannot be constructed by a single traversal of the cycle.

The approach to coverage is to exercise each predicate in the program at least once to each of its possible outcomes, but not necessarily in every possible combination.

The research at GRC in this area concentrated on tradeoffs between segment size and number, and in learning the effects in execution time requirements both in performing the computations and directing the backtracker.

TRW

Hoffman[13,14,15,16] of TRW, Houston, is developing the Automatic Test Data Generator (ATDG) for FORTRAN programs. This tool is currently capable of constructing paths through the unit of software and of identifying a characteristic set of paths required to exercise all logical decisions.

The goal of ATDG is to exercise each transfer at least once using the fewest number of cases.

Hoffman identifies all segments of a program by number, identifies all possible logical transfers between the segments and then defines a path through the software which is a chain of transfers beginning at an entry point and ending at an exit point of the unit of software.

He accomplishes the path determination by constructing a directed graph with each segment of the program represented as a node in the graph and each logical transfer as a node connection, then performing a network analysis to describe the paths. Transfers are the connections between two segments and a segment ends at a predicate with the next segment beginning with the associated executable part of the decision statement.

Unexecutable paths are eliminated using the "impossible pairs" technique. This technique identifies impossible pairs of transfers, based on data constraints that prevent a pair of transfers from being executed in the same pass through the software.

In the majority of cases, the pairs are always impossible. However, if values of variables can be changed, then it is necessary to qualify the possibility of transferring.

The application of the impossible pairs technique to a short program illustrated in (15) reduced the number of possible paths from 3264 to 9. This was further reduced to seven. To reduce the number of paths required to exercise all transfers, a connectivity matrix is generated representing the direct transfers in the software unit. A reachability matrix is then formed allowing a look-ahead capability to determine the best transfer to select based on the number of transfers which can be subsequently executed. Once executed, a transfer is flagged and each new path contains the largest number of new transfers. This process continues until all transfers are exercised.

ATDG does not currently generate data to exercise the paths. The work is continuing and the expansion of the connectivity matrix concept is being pursued. The basic difference between Hoffman's approach and that taken by Howden and Miller is the method of determining path construction. Hoffman defines the paths based on the data constraints, and displays paths which can be executed. This system is interactive with the user generating the test data which will exercise the path.

Hoffman's use of the software network analysis with the connectivity matrices allows the combination of the structural and logical characteristics of the unit to form a matrix defining executable paths based on data constraints. Loops are considered to require one iteration the first pass through the loop bumping the index and the second pass allowed to fall through.

An obvious advantage in the concept of connectivity matrices is the ability to handle units of code that are complex. The size of the matrices increases with the complexity of the program, but the network analysis and matrix generation and processing remains the same.

University of Colorado

L. Clarke[18] of the University of Colorado describes the system developed there to generate test data for programs written in ANSI FORTRAN. The system is an extension to the validation program, DAVE[19], which performs data flow analysis.

Programs are decomposed into segments, and a directed graph is created defining the possible paths. A data base is created containing information about the program units. This data is used during data generation.

The lexical analysis performed creates a list of tokens subsequently translated to an intermediate code similar to assembly language. This intermediate code in conjunction with the directed graph is accessed by the test data generator. An advantage to this approach is that adaptation to a new language would require a translation to the intermediate code, with only minor modifications necessary to create a test generation system for the new language. Howden uses a similar technique.

This system addresses loops and subroutines calls. The user can designate the path to be taken through the program, requesting that loops be traversed a specified number of times and that the path enter and exit specified subroutines following a designated path. The control structure blocks defined by DAVE are numbered and stored in the data base. The paths are then described by the user in terms of the subroutine names and block numbers. Ms. Clarke offers the following example. A path is dezcribed by SUB1, 1,2,5, SUB2, 1,7,8, SUB1, (6,7) $2, EOP,END. An analysis of the path starts with subprogram SUB1. Blocks 1, 2 and 5 are symbolically executed. A call to subprogram SUB2 is encountered in block 5. Blocks 1, 7 and 8 in SUB2 are then executed. A return statement is encountered in block 8 of SUB2 and analysis returns to SUB1, block 5. The remaining code in block 5 is executed. Then the loop formed by blocks 6 and 7 is executed twice.

The analysis determines the feasibility of the path. If it is found to be infeasible, the user is notified and analysis of the path is terminated. If the end of the path is encountered, the path is executable, therefore feasible, and the test data that drove the execution down that path is returned to the user.

Feasibility is determined by attempting the symbolic execution of a path. When a conditional branch is encountered a data constraint is generated. Each constraint is then passed to an inequality solver that attempts to find a solution to the set of constraints and confirm that they are consistent. If a solution exists, the symbolic execution of the path continues. If they are inconsistent, the user is notified and the path is considered infeasible. Linear programming techniques are used to solve the system, based on the premise that a large proportion of programs have linear constraints (allowing the use of the linear techniques) and non-linearities will not limit the use of the tools.

This system operates also in the static mode, with a path being initially specified.

Since the capability of traversing loops creates the possibility of an infinite number of paths, only a few of which may be of interest, this analysis program requires that the path to be analyzed be specified by the user.

IBM

EFFIGY is an interactive symbolic executor developed by King, et.al.[20] at IBM. It is applied to programs written in a special subset of PL/I containing only integer-valued variables and vectors.

EFFIGY accepts one statement at a time, building a tree that defines the paths through the program as it goes along. At each branch, the user decides which path is of interest and communicates this to the system which then proceeds with the symbolic execution.

The path generation proceeds in a forward fashion as opposed to the back-tracking methods used by Howden and Miller.

EFFIGY saves the state of the branches and allows restoration to particular points so that alternative paths can be explored later.

An interesting aspect of EFFIGY is the capability to accept assertions during the symbolic execution that allow comparison of the correctness specification to the data at a given point in the execution.

The use of symbolic input data allows verification of the program over a range of numeric data without examining each specific possible input within that range. It represents another attempt to prove taht data of interest exists that will drive execution down a particular path.

Automation of the process of exploring all paths of interest is being pursued by the developers.

This system is limited in practical use, but gives added insight into the concepts of symbolic execution as a tool in proving programs correct.

Stanford Research Institute

SELECT is a system developed at Stanford Research Institute by Boyer, Elspas, and Levitt[21]. It is also a symbolic executor, executing the code in a forward direction through the program. It is applied to programs that are written in a language that resembles a subset of LISP, since that was the language used as input to the SRI program verifier which was used as part of SELECT.

As the statements are executed and branches are encountered, the path is generated and stored as a conjunction of predicates in a list. Variables are kept in another list which is updated whenever an assignment statement is encountered. Data that will drive execution down the path is also maintained and is derived by the execution of an inequality solver.

As each predicate is examined, the inequality solver is called. It solves linear and some non-linear inequalities for both integer and real valued variables. If the inequality cannot be solved, the traversal of that path is abandoned and is tagged as an impossible path.

Loops are addressed by SELECT as paths that are iterated a user-specified number of times.

SELECT also offers the capability of accepting user-supplied assertions. They can be used to determine the numerical value of the symbolic data during execution, to constrain the input space bounds from which SELECT is to generate test data, or to provide a specification of the intent of the program to verify a given path.

Calls to subroutines are presently handled by substituting the subroutine code into the program. This causes an explosion in the number of paths and is a limitation. Work is being done to address subroutine calls by characterizing them by input and output assertions, and replacing the calls by these specifications.

All test case generators currently described are experimental tools. The various techniques employed in developing these tools represent attempts to achieve a feasible testing aid. All require total knowledge of the program structure as well as the intent, and all are severely limited in the size of the program that can be accepted.

The capability of DISSECT, SELECT and EFFIGY to accept assertions is an added step toward proving the correctness of a program. It allows the programmer to express his testing requirements in some way that reflects the program's intent, and to compare this intent with the results of data derived by the structural analysis of the program.

Execution Analyzers

General Overview

The basic function of execution analyzers is to gather run time statistics that can give a programmer insight into the behavior of this program.

Initial attempts to analyze program behavior were through the use of trace functions. While tracing indeed shows the order and state of execution, it is very costly in programs that are not small, and particularly if loops are involved. Tracing is still a valuable debugging aid since it provides for a display of the contents of cells of interest during execution.

A number of effective execution analyzers have been developed to perform dynamic analysis of the code through the use of external performance monitoring techniques or with software probes inserted into the source code.

Boole and Babbage's Problem Program Evaluator (PPE)[22] is an example of a tool that monitors execution performance externally. PPE is linked to the load module and causes execution to be interrupted periodically, recording the state of the program's execution at each interrupt. The information is compiled and displayed graphically. While this technique does not disturb the test object code and is language independent because it operates at the object code level, it does not provide complete statistics about execution frequency, particularly in tight loops.

The analyzers that insert probes into the source code are language dependent. Among the currently available tools developed for the analysis of FORTRAN programs are MDAC's Program Evaluator and Tester (PET), TRW's Flow program

which is one of the tools in their Product Assurance Confidence Evaluator System (PACE), CSA's QUALIFIER, GRC's RXVP, CAPEX's FORTUNE, and the NBS Analyzer. COBOL analyzers include ADR's Meta Cobol, NCI's Series-J, and CAPEX's COTUNE. MDAC has developed a system to analyze the execution of TACPOL/MOL programs for the Army at Fort Belvoir and is currently working on a prototype PL/I analyzer, and GRC is developing a JOVIAL analyzer for RADC.

Several techniques have been used to accomplish instrumentation of source code. All are based upon the recognition of the program's control structure . Entry points, exit points, branches and loops must be recognized to determine where the probes should be inserted to give the desired results. Most of the tools provide a control language that allows the programmer to communicate with the tool expressing his areas of interest, such as instrumenting only selected code segments and requesting optional statistics. The control language may be transparent to the compiler (in the form of comments) or may be implemented as additional verbs which must be removed after testing is completed.

In either case, the source code, with any embedded control statements is passed through a preprocessor. The control statements direct the instrumenting process (in most cases, default conditions are used when no control statements are present). The preprocessor generates a modified version of the source code that contains the instrumentation. The probes are either calls to subroutines in which the statistics gathering takes place or are counters in the form of assignment statements that collect the statistics in arrays and symbol tables. The performance of the instrumented code is degraded by the added code, but the functional results remain the same (i.e. the program still does what it did before it was instrumented).

Descriptions of the individual tools along with performance statistics for each were gathered during this study (see Appendix A).

All of these tools currently do not actually verify the program. Their value lies in the insight they provide to the programmer about the behaviour of the code in relation to its structure. Dead code can be located, impossible branches can be identified, and high utilization code can be isolated and optimized.

An initial attempt to address the function as well as the structure of the code was proposed by Stucki and Foshee[4] as an extension to PET, MDAC's execution analyzer. A prototype PL/I implementation is currently being tested.

As PET is now implemented, it provides information about the variables in the program as it was executed; namely minimum and maximum values assigned to the variables at each assignment and for each do-loop index variable, and first and last values for variables in each assignment statement.

Current Research

Assertion Checking - General Philosophy

The assertion concepts for programming languages which are now being developed constitute major extensions to our ability to carry out "systematic programming". These new assertion concepts will impact all phases of the software life cycle from initial requirements and design phases down through certification, and maintenance iterations. These assertion concepts are designed to encourage the development of algorithmic validation criteria as the implementation evolves from the initial algorithm requirements and specifications down to the final program code.

The impact of these assertions will be both psychological and real. They will be "psychological" in the sense that they are omnipresent. Embedded as comments within the program code, they will be a positive influence for increased understanding and awareness of how our algorithms should behave and how we plan to insure that our algorithms do in actuality behave properly at all times. The impact of the assertions will be "real" in that they can be automatically checked and monitored under dynamic program execution.

The assertion capability will allow programmers to verify to some extent that the code performs properly within the constraints defined by the assertions. Critical parameters can be dynamically checked or monitored for range, value, and order violations based on the prescribed bounds of the assertions. Subscripts are checked for range violations, and given subroutine parameters are checked for change of value during execution of the subroutine.

The inclusion of specifications data in the program now begins to relate the code to the requirements and is a significant advance toward verifying the program. Module interfaces can be examined via the calling parameter assertions, providing added confidence in the integration of modules.

An important side effect of the assertion capability is the documentation in the code of the critical requirements of the program by the assertion statements. This enhances the understanding and maintenance of the program at all levels of development and operation.

Application of these techniques and their associated tools offers a positive step forward in the development of more reliable software systems. This approach can be applied to existing programming languages today via extensions to currently existing automated tools like MDAC's PET system. The approach also promises to impact future language and language processor design.

Basic Properties of Assertions

The premise upon which the assertion concepts is based is the need we have for thinking through and thoroughly understanding the expected and actual behavior of algorithms. The emphasis will not be placed on proving mathematical or logical properties of programs but rather on an attempt to increase our understanding of the nature and behavior of the algorithms we use. It is openly acknowledged that some purists (i.e., London, Good, et.al.) may feel that we are taking a rather informal approach to the study of program properties. However, it should be noted that the assertion language which will be developed contains sufficient power to serve as a vehicle for stating many formal properties of algorithms.[20] Indeed, at some future point in time a theorem proving tool may well interface with our embedded assertion language.

The assertion language will address both our understanding of flow of control and flow of data through algorithms. A hierarchy of assertion constructs will be defined to make their use more natural and convenient. It should be noted, however, that one assertion construct is really sufficient (i.e., a generalized local assertion).

Generalized Local Assertion

A generalized local assertion may be embedded in a comment at any point within the executable code of a program where another executable statement may appear. The local assertion is designed to enhance the documentation of critical algorithms throughout the entire life cycle of the software. Dynamic execution time checks can be activated at selected points in time to ensure that the actual run-time environment is consistent with the logical state specified in the assertion. This dynamic assertion checking can be used to great benefit in debugging, validation, certification, and maintenance of complex systems.

The format of the generalized local assertion is:

ASSERT LOCAL(extended-logical-expression)[optional-qualifiers]
[control-options]

The exact placement and treatment of the assertions will be tailored to the existing language facilities in currently defined languages. In these currently available languages the assertions will be implemented through specialized comments processed by a source code preprocessor. New language development and future compilers for existing languages may contain options for directly implementing the assertions.

Optional-Qualifiers

In order to provide an existential and universal qualifier notion to the generalized local assertion an optional looping capability is defined: ...FOR $\left|\left|\frac{\text{ALL}}{\text{SOME}}\right|\right|$ $\overset{n}{1}$[(variable-list) (set of ranges/values)]

WHERE

(quantifier-controlling-logical-expr)

e.g. C ASSERT (X(I)¬= X(J)) FOR ALL (I,J) (1:8) WHERE (I¬= J)

means: [∀I,J s.t. 1≤I, J≤8∧ I ≠ J]→ X(I)≠X(J)

Assertion Control Options

The total control alluded to above (i.e., ignoring all assertions by treating them as comments) offers the user a binary choice as to whether or not to apply dynamic assertions during program development, however, other levels of control are provided within the assertion language itself.

The assertion language itslef contains three hierarchical levels of control:

1) instrumentation control - control of those sets of assertions which will be instrumented at a given level of testing.

2) dynamic control - run-time control of those instrumented assertions which are to be dynamiclaly checked, and

3) threshold control - user control when assertion violations are observed.

Instrumentation control is provided by a LEVEL option. The syntax of this option is:

...LEVEL (preprocessor-control-expression)

The LEVEL option provides control information to the preprocessor telling it which sets of assertions should be considered for dynamic analysis. This level of control provides a means for testing selected software features at various points within the software development cycle and fits in well with the top down approach to program development. This also allows a user to group sets of assertions together for various types of dynamic checks.

Dynamic control is provided by a condition option. The syntax of this option is:

...CONDITION(dynamic-control-expression)

The CONDITION option provides run-time control of the assertions which have been built into the program. This option affects only those assertions which have been actually instrumented, thus the CONDITION option is of lower priority than the LEVEL option. It should also be noted that the CONDITION option can be dynamically changed under program control to activate or deactivate the assertion as often as desired.

Threshold control is provided by a LIMIT option. The syntax of this option is:

...LIMIT n[VIOLATIONS] [[HALT | EXIT [VIA] proc-name]]

The LIMIT option provides user control in the event of n violations of the corresponding assertion. The user can specify that control being transferred to a wrap-up procedure proc-name if the EXIT phrase is specified. Otherwise, the HALT phrase will simply terminate execution and generated an assertion report automatically if n assertion violations are encountered. Motivated by a need to make assertions about arrays as well as scalars, the following notation has been adopted.

Array Notation for Assertions

Two areas of concern immediately arise when discussing data arrays, namely, array indices and array values. Thus, if one is monitoring program behavior, it is not enough to monitor array values alone, since program logic is invariably concerned with where these values are stored within the array.

The approach is to generalize the assertion and monitor capabilities to include data arrays. Array notation is as follows:

Assume an array of the form $A(I_1,I_2,I_3,\ldots I_n)$. References to specific subsets of array values or array indices are indicated by $A(I_1,I_2,I_3,\ldots I_n)$, where I_i is a subrange of I_i. This notation is position dependent; i.e., if I_2 is not referenced, its position must be indicated by an asterisk (*), as in $A(I_1,*,I_3\ldots I_n)$. The format of each I_i is $\ell:\mu$ where $\ell<\mu<I_i$ (see note below). If $\ell=\mu$ $\ell:\mu$ may be replaced by μ, as $A(\ell_1:\mu_1,\mu_4,\ldots)$. Thus for A(10,20) we might reference

```
A(5,10:15)
A(*,3)
A(2:5,*)
A(2:6,2:10)
```

Extended Logical Expressions

Two types of extended local operations have initially been defined for the assertion language. An array to scalar logical operation will be allowed with its result being defined as 'true' if and only if all component to scalar operations are found to be true. An array to array logical operation will be allowed for identically specified array cross-sections with its result being defined to be true if and only if each pairwise component operation yields a result of true.

Note: If the character ':' (colon) is not available on a specific machine, another suitable character can be substituted.

Local Assertion Examples

A simple local assertion example is shown below in a typical report format. The assertion simply indicates that at the point where it is inserted into the source code we expect the value of the variable MOVE to be less than 9. The report format indicates that this assertion was checked 9 times. Violations were noted on the 6th and 7th executions of the assertion. It is furthermore noted that MOVE actually contained the value 9 on those two instances. A snapshot is taken of all pertinent variable values associated with the violation when the trace mode is specified.

```
                                        EXECUTION
                                          COUNT ┐     SPECIFIC EXECUTION DATA
                                                ▼
00045   C   ASSERT LOCAL (MOVE .LT. 9) LIMIT 10  9   ASSERTION VIOLATIONS     2

                                                  EXEC NUMBER   VARIABLE VALUE
                                                      6          MOVE    9
                                                      7          MOVE    9
```

It is also worth noting that had we encountered 10 violations we would have halted execution at this point in the program.

Examples of the use of array cross-sections in extended logical expressions include the following: (assume an array A(10,20) has been defined)

(a) ASSERT LOCAL (A(*,3) .LT. 10)
LIMIT 6 VIOLATIONS

(b) ASSERT LOCAL(A(2:6,2:10) .NE. 0)

(c) ASSERT LOCAL (A(*,*) .GT. 0)

In (a), the value of each array element whose second subscript=3 is checked and reported as a violation if its value is not less than 10. Ten array values will be checked in all. Any number of assertion violations within an array operation cause the operation to be counted as a single assertion violation. Thus, the LIMIT 6 VIOLATIONS is concerned with only the number of invalid operations not with the number of violations within the array.

In (b), only array values within the specified subscript ranges are checked for an assertion violation. In (c) all array values are checked for an assertion violation.

Specialized Local Assertions

A number of additional specialized local assertions are proposed to facilitate the expression of user validation criteria. This extensible attribute of the local assertion concept is illustrated by the following constructs:

```
ASSERT LOCAL VALUE[S] (variable-list) (set-of-legal ranges/values)
   [control-options]

ASSERT LOCAL VALUE[S] (variable-list) NOT (set-of-illegal ranges/values)...

ASSERT LOCAL VALUE[S] (variable-list) INVARIANT...

ASSERT LOCAL SUBSCRIPT RANGE (list-of-array-specification)...

ASSERT LOCAL ORDER (array-cross-section) [ ASCENDING | DESCENDING ]...
```

All of these specialized local assertions could be replaced by one or more generalized local assertions, however, their existence facilitates the graceful transition from program requirements and their associated validation criteria to embedded program documentation in the evolving code.

The first three constructs cause instrumentation to be generated at the position where they occur. The latter two constructs specify that the next executable statement will respectively be checked to insure that it does not alter the value of an invariant variable (e.g., through side effects from subroutine or function calls) or use subscripts outside the specified ranges.

The ASSERT ORDER statement checks a sequence of array values as follows:

```
ASSERT LOCAL ORDER (A(*,3)) ASCENDING
```

For an array A(10,20), the following assertion violation summary illustrates the type of information traced for a violation:

```
EXECUTION
COUNT          SPECIFIC EXECUTION DATA

 229       ASSERTION VIOLATIONS              1
            EXEC NUMBER SEQUENCE SNAPSHOT VALUE
                18        A(7,3)          6
                          A(8,3)         100*
                          A(9,3)          8
```

The ASSERT VALUES statement checks variable values against a specific set of ranges/values. The ASSERT SUBSCRIPT RANGE statement will check addressing on those arrays specified to ensure that only those portions of the array specifically selected are accessed. A subsequent example will illustrate the usefulness of this concept later in this paper. All of these latter constructions will result in providing similar traces to those already presented for out of bound conditions.

The Concept of a Global Assertion

Expanding our notion of assertions, we immediately identify the need to expand to scope of application for our asserted program properties. In an effort to avoid requiring several similar local assertions within a particular program region, the concept of a global assertion has been introduced. This is a novel approach which promises to have a significant impact on the way we design, implement, and test software.

Global assertions will allow us to extend our capacity to inspect certain behavioral patterns for entire program modules, selected regions of modules or module interfaces (entries and/or exits). Global assertions appear in the declaration section of the program module.

These global assertions will have effect within the scope defined (i.e., globally at all pertinent points, regionally over the named region, collectively for all entries, and/or collectively for all exits.)

The VALUES statement inspects each specified variable as its value changes and reports when: (option 1) the new value is not one of the specified legal ranges/values, or (option 2) the new value assumes a specified illegal ranges/values, or (option 3) checks to make sure the values of the selected variables are preserved (i.e., no direct or externally caused changes are permitted).

The ASSERT SUBSCRIPT RANGE statement verifies that array subscripts fall within a specified range whenever the array is referenced during program execution. It should be noted that this statement provides a means of checking portions of arrays as well as normal upper and lower bounds. For this reason, it is more powerful than the PL/I type ON SUBSCRIPT RANGE check.

Instrumentation will be inserted into the source program by the preprocessor to accumulate the following statistics relative to assertion violations:

(1) Identify the statement that caused the assertion violation. For that statement an execution count and violation execution counts identical to those obtained for local assertions are reported.

(2) The actual value that caused the violation. This value is linked to the statistics identified in (1) above.

Some FORTRAN examples follow:

```
                    .
                    .
20                  DIMENSION A(10,20)
21        C         GLOBAL TRACE 10 VIOLATIONS
22        C         ASSERT VALUES(I,J,K,L) (0:100)
23        C         ASSERT VALUES(II,LL) (-10:10)
24        C         ASSERT VALUES (KK,NN) (2,4,6,8,10)
25        C         ASSERT SUBSCRIPT RANGE (A(*,31))
26        C         ASSERT VALUES (X,Y,Z) INVARIANT

                    .
                    .
                    .
102                 K = K + 1
103                 II = A(L,J) + LL
                    .
                    .
                    .
234                 K = A(J,K) + I*100
235                 II = II + 2
236                 NN = KK*(I-J)
                    .
                    .
                    .
300                 CALL ROUTINEX(X,Y)
                    .
                    .
                    .
```

If assertion violations occurred in this example, the following statistics are indicative of what would be reported by the Postprocessor:

	Annotated Program Listing	Execution Count	Specific Execution Data	
102	K = K + 1	511	ASSERTION VIOLATIONS	1
			ASSERT (I,J,K,L) (0,100)	
			EXEC NUMBER VALUE	
			10 101	
103	II = A(L,J) + LL	511	ASSERTION VIOLATIONS	3
			ASSERT (II,LL) (-10,10)	
			EXEC NUMBER VALUE	
			22 20	
			ASSERT SUBSCRIPT RANGE(A(*,3))	
			EXEC NUMBER VALUE	
			5 A(12,3)	
			105 A(1,4)	

```
234  K = A(J,K) + I*100          125     ASSERTION VIOLATIONS                4
                                         ASSERT (I,J,K,L) (0,100)
                                           EXEC NUMBER      VALUE
                                                52            101
                                                53            102
                                         ASSERT SUBSCRIPT RANGE(A(*,3))
                                           EXEC NUMBER      VALUE
                                                52          A(5,4)
                                                53          A(6,4)

235  II = II + 2                 125     ASSERTION VIOLATIONS                1
                                         ASSERT VALUE (II,LL) (-10,10)
                                           EXEC NUMBER      VALUE
                                                50            12

236  NN = (X*(I-J)                38      ASSERTION VIOLATIONS                1
                                         ASSERT VALUES (KK,NN) (2,4,6,8,10)
                                           EXEC NUMBER      VALUE
                                              20             7

300  CALL ROUTINEX(X,Y)           53      ASSERTION VIOLATIONS                1
                                         ASSERT VALUES (X,Y,Z) INVARIANT
                                                    VALUE OF CALL PARM X
                                           EXEC NUMBER BEFORE CALL  AFTER CALL
                                              30          -10          -20
```

Structural and Static Analyzers

Structural analyzers are tools that examine the program code, performing an analysis of the structure. Execution of the code is not required for this analysis.

The analysis results in definition of the internal control structure describing the paths through the program and can be depicted as a directed graph. Other analyses can be performed based on the decomposition of the structure into a tree representation.

Structural analysis is performed by JOYCE(MDAC), RXVP(GRC)[23], PACE(TRW)[24] and both BRNANL and DAVE (University of Colorado)[19,25]. These systems all provide an error finding capability at this level, and in addition, are the foundations of the test case generators developed by each organization. They all are designed for analysis of FORTRAN programs.

Hoffman's Automated Test Data Generator (ATDG) uses the structural analyzer component for PACE. Miller's Automated Test Case Generator uses the structural analyzer component of RXVP. Clarke's test case generator uses the structural analyzer, DAVE. The decomposition methodology is described in the discussions of the respective test case generators.

The analysis of a program's structure also provides a foundation for various types of static analysis, particularly the relationships and data flows within programs.

DAVE examines a program consisting of one or more routines, and checks for a number of common errors not detected by compilers. Its philosophy is based on two rules expected to be obeyed in the execution of a program, 1) that a variable is not referenced unless previously defined, and 2) that once defined, it is subsequently referenced, before being redefined or undefined. The principle here is that many common programming errors cause these rules to be violated. Therefore, a search for violation of the rules should reveal the errors or possibility for errors that cause the violations.

Among the errors that can cause violations of the rules are uninitialized variables, misspelling of variable names, unequal lengths of corresponding argument and parameter lists, and mismatched types and dimensions of arguments and parameters.

DAVE constructs a call graph which represents the relationships between subprograms being called. The subprograms that do not invoke other subprograms are called leaf subprograms and analysis of the code begins with them.

In order to detect the rule violations, it is necessary to know the usage of every variable in every statement, (i.e. whether it is used as input or output for each case). Therefore, a search of the subprograms is conducted along the paths defined by the structural analysis, to look for the rule violations. DAVE has addressed the problem of data passing through calling parameters and through COMMON, and provides information about these variables in terms of their input/output classification.

Much of the analysis performed by DAVE is not designed to identify specific errors but to sense suspicious situations where errors commonly lurk, and to pass warnings to the user who must then determine whether or not an error actually exists, or whether the program can be improved.

The static analysis provided by RXVP provides much the same capability, but with added features such as statement classification by type of statement, number in each module and percentage of total (note that this is also a capability of the execution analyzers); a module cross reference table of variables, the statements that reference them, and their type of usage; and an array subscript check for indexing appropriate to the array definition.

TRW's PACE program incorporates some static analysis capabilities into its structural analysis and execution analysis program.

JOYCE is an automatic checkout and documentation aid. It compiles tables of symbols and cross references of symbol usage within each routine of a program. These symbols include FORTRAN variable names the names of any referenced function or module, any entry points, and all I/O file references. JOYCE permits the input of symbol descriptions as data. This information may describe or designate a variable definition, a math symbol, flags for grouping related subsets, or subroutine usage information.

The edited information may be combined to produce several combinations of descriptive reports.

The cross-reference lists are useful for verifying consistency of symbol naming and usage, for finding typographical errors in coding and checking a program's logic flow.

Subroutine flowlists aid in verifying the accuracy of the modules logic flow.

It is impossible to separate the tools strictly by function. There is a great deal of overlap in the types of analysis performed by each tool. Seldom does a tool serve one function, therefore, it is difficult to evaluate or even discuss them solely within one given category. For example, MDAC's PET program, while basically classified as an execution analyzer, also performs static analysis in the form of providing a syntactic profile giving the statement types and number of each. Again, the syntactic profile does not in itself identify specific errors, but provides information useful in verifying that certain properties of the program exist (e.g. ample commentary) or that certain violations do not exist (e.g. non-standard statements).

Test File Managers and Generators

A test file generator differs from a test case generator in that it generates data based solely on input parameters specified by the programmer rather than on any knowledge of the program. Its purpose is not to try to prove anything about a program but to relieve the programmer (or test personnel) of the tedium of manually generating large volumes of data.

Test file managers are tools that allow easy manipulation of test files once they are generated.

These tools are commercially available for COBOL applications where they are particularly useful in testing systems that are data base driven. Among those offered are PRO/TEST (Synergetics)[26], Series-J (NCI)[27], and MetaCOBOL (ADR)[28].

PRO/TEST is a set of three tools, the Test Data Generator, File PRocessor and File Checker.

The test data generator generates data based on parameters specified on input cards. These parameters specify the file structure (record and field definition), the data ranges, conditional operations such as generation based on comparisons, and computational operations which derive data by operating on data from two data fields or one data field and a constant. Random specification always causes generation of the same data from the same parameters.

The PRO/TEST file processor provides the capability of coupling live data with generated data, allowing selection and editing of records.

The File Checker matches the output of a program to the original record design specifications to verify that the format and the structure are correct.

Series-J offers a test data generation capability by parameter specification internally. It provides a Test Division in the code that specifies the data and its format. It then generates programs that will generate the data from tables within the generated programs.

The test data generator generates data based on parameters specified on input cards. These parameters specify the file structure (record and field definition), the data ranges, conditional operations such as generation based on comparisons, and computational operations which derive data by operating on data from two data fields or one data field and a constant. Random specification always causes generation of the same data from the same parameters.

The PRO/TEST File Processor provides the capability of coupling live data with generated data, allowing selection and editing of records.

The File Checker matches the output of a program to the original record design specifications to verify that the format and structure are correct.

The MetaCOBOL system includes a test data generator that also accepts input parameter specifications, generating data as specified.

Other tools of this type include MDAC's random data generator which uses the computer's internal clock as the seed for the random number generator, and then builds a test data file within the bounds of input parameter specifications. This tool generates numeric data as opposed to structural data generally output by the COBOL data generators, and is currently used to test FORTRAN programs.

TRW has two tools called ATC and RETEST that are test file managers for FORTRAN systems. ATC provides the capability to store pre-defined test cases in the data base, edit these test cases, and selectively execute the test cases. It also assists in the automatic comparison of test output against previously generated output.

RETEST is a tool used to identify test cases required to reverify software that was previously verified and to identify new cases where required because of coding changes.

CSA offers commercially a tool named RETEST which is similar to TRW's RETEST.

Miscellaneous Tools

There are a number of other tools that can aid in program testing. Most of these are application dependent and must be redeveloped for every application.

However, there are a few that provide valuable information and are not application-dependent. MDAC's JOYCE is an example.

Probably the most commonly used type of tool is a simulator. While there are myriad simulators in existence, the requirements upon them are usually highly specialized. Therefore, there are not any generalized simulators in common use. The two simulators offered commercially, SAM (ADR) and CASE (TESDATA) are management visibility tools rather than actual test tools, and therefore will not be discussed here.

There are many types of tools which have a common concept behind their design but which are application dependent due to the need to use data specifically unique to the system being tested. These tools include data verifiers, event loggers and their associated deloggers, and performance analyzers.

There are several tools available whose utility is strictly in the debugging area. While there is often a fine line between debugging and development testing, this study will not address those tools that can be defined as debugging aids.

Conclusions

This task surveyed many of the currently available program testing tools from the viewpoint of philosophy as opposed to performance which was addressed in Task I (see Appendix A).

The application-independent tools in use at present do not verify that the code in any way meets the specifications for which it was written. They give insight into the structure and behavior of the code. However, in light of the size and complexity of today's computerized systems, any tool that relieves the programmer and the tester of time-consuming, tedius, and error-prone work is a valuable asset in the production of reliable software.

One new and very promising area of research now being explored addresses techniques for using tools to check functional attributes of programs. Through the dynamic checking of asserted program properties testing tools can provide valuable feedback on program compliance with selected specifications.

As research continues it appears that the concept of proving that a program reliably solves the problem that was intended will eventually become a reality. In today's environment, where practicality is a prime issue, the tools that can provide the most useful help to the user are the standards checkers, the execution analyzers (provided the language and machine attributes match), the test data generators and managers, and the structural analyzers. Other tools such as test case generators like DISSECT are still in the experimental stage of development and will require more research before they reach the stage of practical usefulness.

In surveying these tools, the most common complaint was the difficulty of use. As new tools are developed and old ones are upgraded, greater attention to user orientation will help make these tools more valuable, simply because they will be used more.

REFERENCES

1. G. M. Weinberg. The Psychology of Computer Programming, Van Nostrand Reinhold Company, New York 1971.

2. F. T. Baker. Chief Programmer Team Management of Production Programming IBM Systems Journal, Volume II, No. 1 (1972) pp. 56-73.

3. Information Research Associates. Reliability Techniques for Computer Executive Programs, Summary Report NAS8-2666-9.

4. L. G. Stucki, G. L. Foshee. New Assertion Concepts for Self-Metric Software Validation. 1975 International Conference on Reliable Software, Los Angeles, California, April 1975.

5. Code Auditor Requirements Specifications, TRW Systems Group, Working paper.

6. The Age of Metacobol. Applied Data Research (ADR), January 1974 Sales Brochure.

7. Standards Auditor. Computer Software Analysis,Inc., Sales Brochure.

8. Ryder, B. G. The FORTRAN Verifier: User's Guide, Bell Laboratories, Computing Science Technical Report #12.

9. Howden, W. E. and J. Laub. Automatic Case Analysis of Programs, Computer Science and Statistics Eight Annual Symposium on the Interface, Los Angeles, California, February 1975.

10. W. E. Howden. Methodology for the Automatic Generation of Program Test Data. Technical Report #41, McDonnell Douglas, February 1974.

11. W. E. Howden, L. G. Stucki. Methodology for the Effective Test Case Selection Final Report MDAC-W, MDCG-5301.

12. W. E. Howden. Proposal to Investigate a Methodology for Effective Test Case Selection. McDonnell Douglas Astronautics Company, March 1975.

13. R. H. Hoffman. Automated Verification System: Test Data Effectiveness Measurement Subsystem User's Guide, NASA/JSC Internal Note No. 74-FM-46, June 1974.

14. R. H. Hoffman. NASA/Johnson Space Center Approach to Automated Test Data Generation. Computer Science and Statistics Eighth Annual Symposium on the Interface, Los Angeles, California, February 1975.

15. W. E. Hoffman. Advanced Techniques in the Generation of Connectivity Matrices for Software Network Analysis. TRW Systems Group, working paper.

16. J. R. Brown, M. Lipow. Testing for Software Reliability. Computer Science and Statistics Eighth Annual Symposium on the Interface, Los Angeles, California, February 1975.

17. E. F. Miller, Jr., R. A. Melton. Automated Generation and Test Case Datasets. International Conference on Reliable Software, Los Angeles, California, April 1975.

18. L. Clarke. A System to Generate Test Data and Symbolically Execute Programs. Report #CU-CS-060-75, February 1975, Department of Computer Science, University of Colorado, Boulder, Colorado.

19. L. J. Osterweil and L. D. Fosdick. Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation and Error Detection. Report SS(September 1974), Department of Computer Science, University of Colorado, Boulder, Colorado.

20. James C. King. A New Approach to Program Testing. 1975 International Conference on Reliable Software, Los Angeles, California, April 1975.

21. Robert S. Boyer, Bernard Elspas, Karl N. Levitt. SELECT -- A Formal System for Testing and Debugging Programs by Symbolic Execution. 1975 International Conference on Reliable Software, Los Angeles, California April 1975.

22. Boole and Babbage, Inc., Product Description Documents Sunnyvale, California.

23. RXVP-1 User's Guide, General Research Corporation, February 1975.

24. J. R. Brown. Practical Applications of Automated Software Tools. Preprint 21/3, 1972 Wescon Technical Papers, September 1972.

25. L. D. Fosdick. BRNANL, A FORTRAN Program to Identify Basic Blocks in FORTRAN Programs, Report #CU-CS-040-74, Department of Computer Science, University of Colorado, March 1974.

26. An Introduction to PRO/TEST, Synergetics Corporation

27. Series-J Summary Description, National Computing Industries.

28. The Age of MetaCOBOL, Applied Data Research, 1974.

Appendix D

STRUCTURED PROGRAMMING AND PROGRAM MANAGEMENT TECHNIQUES

D.1 INTRODUCTION TO STRUCTURED PROGRAMMING

The phrase "structured programming" probably had its origin in Dijkstra's "Notes on Structured Programming" which were privately circulated prior to being published [Reference 1]. In these "Notes", Dijkstra's main concerns are the problems of very large programs and the methods by which their reliability can be improved. At least three major ideas are present.

1. There is a need for some sort of a "demonstration of a program's correctness" to supplement the standard functions of design code and test.

2. Programs should be coded using only three types of control structures.

3. Programs should be composed in a top-down manner utilizing systematic design techniques.

Taken together, these ideas are revolutionary even though all three were developed to some degree prior to Dijkstra's Notes. His major contribution was to bring these three diverse ideas together with a convincing argument that only in this manner could reliable software be developed.

Dijkstra apparently feels that there is a deficiency in the standard cycle of design, code and test and that some sort of "demonstration of a programs correctness" should be included in this cycle in the future. He points out that program testing is an imperfect answer to the problem since "Program testing can be used to show the presence of bugs, but never their absence". This idea of demonstrating program correctness is being seriously considered in the more practical oriented segments of the computer industry but to date it has been primarily a subject for research and development.

In its pure form, the method of "demonstrating program correctness" which has evolved, is very similar to the proof of a theorem in theoretical mathematics. Both manual and automated methods of performing such "program proving" have been studied intensively.

A pertinent question at the present stage of development is whether some form of useful demonstration of program correctness is possible short of the very sophisticated techniques of program proving. For example, a reasonably rigorous discussion of key facets of an algorithms' mathematical and/or logical basis could easily be imposed as a minimum requirement. Such information is often available in a document describing the algorithm development but usually this information does not impact the coding phase

and more importantly, does not form the basis for any system testing. Perhaps algorithm code should be designed not for maximum efficiency but rather for maximum clarity of these key mathematical and/or logical bases. The results of research in the program proving field suggest examples of the types of things which should perhaps be emphasized in both coding and testing.

Non-Trivial Loops

Some reassurance should be given that loops terminate properly under all possible conditions. The parameters that control looping should be identified and the exact manner in which these parameters change should be explicitly stated. Some reassurance should be given that the loop will be executed the correct number of times.

Invariants

An invariant is a relation which is true at every iteration of a loop. A clear and concise demonstration that a loop truly accomplishes what it is designed to do is often most easily performed by identifying an invariant associated with it. [Reference 2 demonstrates the use of invariants].

Decision Structures

Some reassurance should be given that all possible alternatives have been accounted for by some path in the decision structure. The conditions under which a particular program path will be taken should be explicitly stated.

The key feature seems to be explicit identification of key parameters and relations. Once this has been accomplished, automated methods can be used to verify that these assertions are valid. For example, software probes can be inserted in the program to be tested to verify that loops are executed the correct number of times, that invariant relationships always remain true, and that a deliberately designed series of tests exercise all paths of a decision structure.

The idea that programs should be coded using only three types of control structures probably dates to a paper which proves a theorem concerning the flowcharts of proper programs. [Reference 3] (A program - or program segment - is proper if it has a single entry point and a single exit point). The theorem states in essence that the flowchart of every proper program can be represented by an equivalent flowchart which is composed of only three basic control structures (Figure 1). Dijkstra goes a step further and suggests that only these three control structures should be used. [In Reference 4, he singles out the GOTO or unconditional branch instruction as a particularly common offender of this coding methodology). In support of this suggestion, Dijkstra presents several compelling arguments:

(1) The intellectual effort necessary to understand such a structured program is roughly proportional to its length (measured in some loose sense). This is not true of unstructured programs whose complexity often increases geometrically with length.

(2) The SEQUENCE and IF-THEN-ELSE control structures can be understood by enumerative reasoning and the DO-WHILE control structure by mathematical induction. Because "we know the appropriate pattern of reasoning", the task of demonstrating a program's correctness is made easier.

(3) In such a structured program, the machine's "progress through the computation is mapped on progress through the text in the most straightforward manner." In other words, the program's execution sequence is more like the instruction sequence written on the listing than for a non-structured program.

The third major idea in Dijkstra's "Notes" is the concept of what has come to be called top-down programming (Dijkstra called this Stepwise Program Composition). As motivation for composing programs in this manner, Dijkstra cites the problems of program modification and program manageability. Other authors including Wirth have also discussed this topic extensively [Reference 5].

The basic approach is to compose a program in minute steps, deciding at each step as little as possible. As the problem analysis proceeds, so does the further refinement of the program. In such a stepwise approach, certain aspects of the problem statement are ignored at the beginning. This judicious postponement of decisions and commitments results in decisions being made at lower levels than perhaps they otherwise might be. One result of this is that program modifications can be made at these lower more isolated levels where their impact is less. More importantly, however, such a program composition is claimed to result in a higher level of abstraction program. When a program has been built-up to an intermediate state of refinement, what has been written down is a suitable "common ancestor" for all possible programs which can be produced by further refinements. In other words, the structure of the program is such as to anticipate its adaptations and modifications. As Dijkstra puts it, "The similarity between program modification and program composition is the similarity between the decision to be changed and the decision still left open".

The starting point of the program composition is a concise statement of all of the things which the program is expected to do (e.g., the highest level program specifications). Thereafter, one proceeds by conceiving a "computation" of "more primitive actions" that accomplish the desired net effect. If these more primitive actions belong to what Dijkstra calls "The well understood repertoire" (e.g., they are computer executable instructions) then one is done. Otherwise, the process continues. At least four things about this approach are desirable from a program management point of view.

(1) The starting point is the program specification which provides maximum management visibility of the program development process from inception.

The highest level requirements which are of prime interest to the program management and customer are addressed at program inception whereas decisions on detailed specifications (which are most subject to change) are delayed to the later stages of the development effort. This, of course, is the reverse of the traditional "bottom-up" development effort.

(2) Much of the early stages of program design can be performed via English language statements. Thus, early versions of the program will consist largely of computer instructions mechanizing the highest levels of control structure and a large number of English language statements (comments) which document in detail the decisions which have already been made and equally important, those still left open. Such a program is self-documenting to a very high degree. Also, there should be a very close correspondence between the program comments and the program specification document.

(3) A running program is in existence virtually from program inception. Integration is accomplished by adding refinements to the existing program in the top-down manner. Thus, integration is a continuous process performed throughout the development cycle. This contrasts strongly with the usual bottom-up development process where integration is the last and often a very traumatic step prior to final testing.

(4) Program testing is also a continuous process performed throughout the development cycle. When refinements are added at any level, the program testing necessary to verify the requirements associated with these refinements is performed. There are three potential benefits of this approach. First, the higher level functional requirements are tested early in the development. This should tend to provide early reassurance to the customer that his most important requirements are being met. Second, testing at any level provides an important additional test of higher level code. Thus, the very important highest levels of code are exhaustively tested since they are exercised to some degree by the testing performed at all lower levels. Finally, the program itself acts as the "driver" for all testing. The need for separate "driver programs" to perform unit tests (as in the bottom-up approach) is eliminated.

Before the reader gets the impression that "top down programming" and related design techniques are the answer to all the world's problems, one final point should be emphasized. This is simply that these design techniques are very difficult, especially the first time they are applied. The temptation to plunge into great detail in the "firm" areas rather than make hard decisions which are required in the "not so firm" areas must be resisted. Also, making the "right" decision is quite difficult. Subsequent developments may show clearly that a particular decision was wrong or

that a particular decision should have been deferred and that a decision on a separate issue should have been made instead. The capability to backup and start again from a higher level must be present. The results of "errors" are out in the open for all to see and as a result no stigma should be attached to the individual who makes a decision which turns out to be wrong. In short, top down programming requires a tremendous change in management practice.

Whether or not this change can be made smoothly is yet to be seen. Probably the outstanding example of top down programming management is the IBM Chief Programmer Team experiment discussed in Section 2.2.2. The published accounts of this experiment claim results which can only be called outstandinh. Tempering all the enthusiasm, however, is the strong probability that the personnel involved in this experiment were of the very highest caliber. It remains to be seen if similar results can be attained by a team of more modest talents. What the Chief Programmer Team experiment may really be saving, is that outstanding personnel when highly motivated will produce outstanding results.

D.2 CURRENT STATUS OF STRUCTURED PROGRAMMING

Structured programming was originally intended as a collection of programming disciplines which have in common the objective of producing reliable software. Over the years, this collection of ideas has fragmented and today each of the major ideas previously discussed has become an entity in itself. The concept of a demonstration of program correctness has become an area of active research but as yet little practical application. The concept of a limited number of control structures has been accepted by a substantial number of people and today it is often this concept that people refer to when they discuss "structured programming". Finally, the concept of top down programming has blossomed into a new software management philosophy of which the IBM Chief Programmer Team is the best known example.

Because these ideas have developed along such separate paths, they are discussed in detail in three separate sections. Since program proving relies heavily on other aspects of "structured programming" it is presented last.

D.2.1 Structured Coding

The idea of coding a program using only a limited number of control structures is simplicity itself. Its theoretical basis is the theorem proved in the classic paper which states that any proper program segment can be flowcharted using only three control structures (Figure 1) [Reference 3]. Of course, the fact that a program fan be so constructed is in no way a valid reason that a program should be so constructed. The arguments of Dijkstra and others have been generally accepted as valid arguments that coding should be restricted to a limited number of control structures, but the number and composition of such a set are subjects upon which there is virtually no agreement.

The main problem stems from the fact that neither the IF-THEN-ELSE nor the DO-WHILE constructs are sufficiently general to satisfy a large number of programmers who are required to write "real world" programs. The IF-THEN-ELSE is not a general decision construct since it allows a choice between only two alternatives. The "case" construct (Figure 2) is a more general decision construct and the other construct shown in the figure is still more general. In a similar manner, the DO-WHILE is not a general loop construct primarily because only a single exit from the loop is allowed. A more general loop construct is illustrated in Figure 3.

A second practical objection to the three control structure limitation is the desired capability to exit from deeply nested code. There are two types of desired capability along these lines which are really quite different. The first is an immediate exit to a specified location in another subroutine. The first capability is easily mechanized with a GOTO statement whereas the second is a much more difficult capability to implement.

Most major languages implement a return to the invoking subroutine (procedure) upon conclusion of the invoked subroutine (procedure) and no significant objection has arisen to including this construct within the realm

of "structured programming". Many languages (e.g., FORTRAN) also implement a return to the invoking subroutine from an interior point of the invoked subroutine. Objections to this construct are almost as common as the objections to the "GOTO". Since no major language includes a more general capability along these lines, the need for it is hard to justify. Some limited capability along these general lines seems reasonable and is often cited as justification for retaining use of the "GOTO" and the "Return".

In theory, code can be structured in any programming language. The ease with which it can be performed, however, varies greatly from language to language. In assembly language, it is necessary to simulate the basic control structures. If a macro facility is available, this can be done quite nicely by providing standard macros which mechanize the IF-THEN-ELSE and the DO-WHILE constructs.

Several higher level languages have sufficient control structures to structure coding without modifying the language in any way. (PL/I, ALGOL and COBOL are examples). Even these languages, however, suffer from a lack of generality of the constructs available and/or a proliferation of optional methods of writing what is in essence the same construct. The FORTRAN language lacks an IF-THEN-ELSE construct so to structure code in FORTRAN one must either construct an IF-THEN-ELSE type construct from more primitive operations (i.e., the FORTRAN IF, CONTINUE and GOTO statements) or use some sort of FORTRAN language extension. A number of such FORTRAN language extensions have been proposed and several are operational. In these language extensions to "permit structured programming in FORTRAN", a proliferation of control structure has occurred. Figure 4 presents a representative cross section of control structures which have been proposed as extensions to FORTRAN.

Included is a flowchart of the construct, a typical "structured coding" including indentation of the code for clarity and the construct as it might be mechanized in pure FORTRAN. Table 1 presents a summary of several existing systems which implement extensions to the FORTRAN language. Table 2 lists several characteristics which would be desirable features of such a system.

The control structures given in Figure 4 fall into 4 categories: decision structures, single exit loops, two exit loops and multi-exit loops.

The inclusion of two exit and multi-exit loops probably requires some justification. A two exit loop is a desirable solution to a very common programming problem; i.e., the processing loop in which one set of computations is to be performed if a certain operation is successful and an entirely different set of computations is to be performed if the operation is unsuccessful. Examples of this would be an iteration which either converged or did not and a search which either turned up a match or did not. Situations of this type can be handled by single exit loops; however, the coding is often awkward. Similar situations exist for which multi-exit loops are useful. These situations are not nearly so common. The primary justification for the multi-exit loop is probably that it presents no more difficulty than the two exit loop. If one accepts two exit loops as a valid control structure for producing structured code, it is difficult to conceive

any reason for excluding the multi-exit loop. Discussions of multi-exit loop control structures are presented in References 6 and 7.

The number of control structures in each category and the diverse ways in which each control structure may be coded give some indication of the problems involved in standardizing the control structures for structured coding. Some of the problems are (1) the concept of a block structure, (2) how to express the general decision control structure [Figure 4,#4] (3) how to express the simple loop structure, and (4) what is the role of index variables.

The imposition of a block structure is undoubtedly motivated by the modern trend to block structured languages. The reasons for this are rather subtle and have to do with the efficient mechanization of certain advanced features (e.g., recursive procedures and advanced data structures such as variable length strings, lists and stacks). The block structure is generally considered to be more desirable than the simpler subroutine structure of FORTRAN for the mechanization of such features (The FORTRAN language does not include such features so the issue really never arises). It remains to be seen whether there is a net advantage in imposing some form of block structure on the FORTRAN language. There are, however, several systems which include this capability in some form.

The fourth control structure in Figure 4 is a very general decision control structure that appears in many structured coding systems. The literature, however, includes at least three different ways of viewing this construct (see the figure) which are suprisingly different. The first viewpoint imposes a block structure such that the code for each alternative is contained within a separate block. The second viewpoint considers the construct to be an extension of the IF-THEN-ELSE statement. The addition of an ORIF or similar statement converts the IF-THEN-ELSE into a general decision structure. The third viewpoint considers the construct as a generalization of the simple CASE statement. The details of this generalization are discussed in the note appended to construct #3 of Figure 4.

The single exit loop structure suffers from a proliferation of optional methods of writing code. There are two main problems: (1) where to place the test for exit from the loop and (2) the role of index variables. The DO-WHILE places the exit test at the beginning of the body of the loop code. This is a more general construct than the DO-UNTIL (which places the test at the end), since the body of the loop can be bypassed (i.e., executed zero times). The DO-UNTIL, however,often produces code which is easier to understand. For example, the code DO LOOP UNTIL I=10 impacts the message that the loop is to be executed ten times somewhat better than the code DO LOOP WHILE I<11. Because these loop constructs fix the point at which the exit test is made, the loop control and termination test can be specified through an index variable (e.g., the variable I in the example code above). This saves the programmer the trouble of writing the code for the loop control parameters and the termination test. More importantly, however, this removes a possible source of coding error. These advantages must be balanced against the disadvantage of lack of generality and flexibility. Construct #7 shows a more general loop in the sense that the exit test can

be anywhere in the body of the loop. The price paid for this generality is that the programmer must define his own loop control parameters and include code for the termination test. This in turn introduces potential sources of error.

Similar constructs are possible for multi-exit loops. For example, constructs #8 and #9 are the two exit equivalents of the DO-WHILE and the DO-UNTIL. Multi-exit loops, however, are sufficiently complicated that the additional complexity introduced by requiring the programmer to code loop control parameters and termination tests seems minor. The general form of #10 would therefore seem more appropriate.

TABLE 1

SOME TOOLS FOR "STRUCTURED PROGRAMMING IN FORTRAN"

Name	Organization	Control Structures (Figure 4)	Index Variable Options	Other Features	Limitations	Status
IFTRAN	General Research	4,5* 1,6**	Programmer's Responsibility			Oper
PREFOR	IBM	1,5,6,7,8,9	Programmer's Responsibility	3 4a,4d,4e		Oper
MORTRAN	Stanford Linear Accelerator Facility	1,5,6	Many forms of DO loop	8		Oper
TRANSFOR	Boeing Computer Services	1,2,3,4,5,6		4a,4d		Oper
SCIF		1,2,4,5,6				
STAPLE	National Bureau of Standards	4,7* 1,5,6**		4a,b,c,d		
PSST	McDonnell Douglas	4,7,10* 1,5,6,8,9,10**	Programmer's Responsibility	1,2,5,6,7		Dev

*Directly Available

**Equivalent capability available as a special case of a more general construct.

Other Features

1. Automatic listing indentation.
2. Preprocessor
3. Compiler
4. Block Structure
   a. BEGIN BLOCKS
   b. SELECT BLOCKS (similar to Case construct)
   c. REPEAT BLOCKS (similar to DO loop)
   d. EXIT BLOCK
   e. EXECUTE BLOCK (similar to Perform verb in COBOL)
5. Free Form Input
6. Accepts Pure FORTRAN
7. Comments on executable statements
8. Source Macros

TABLE 2

DESIRABLE CHARACTERISTICS OF STRUCTURED PROGRAMMING AIDS FOR FORTRAN

1. Easy to learn. A minimum number of additional control structures. There seems to be general agreement that at least two are required (e.g., an IF--THEN-ELSE type construct and a loop structure which does not require statement numbers).

2. Mechanized indentation of listings.

3. Free Form code input. In particular, should accept code in the indented form (such as presented on the listing) but not require it.

4. No change to the FORTRAN language - only additions. In particular, a pure FORTRAN program should pass through the preprocessor unchanged and execute properly.

5. Capability to include a comment on each statement for self-documentation purposes.

6. Full user control of Listing Format, delimiters, etc.

D.2.2 Program Management Techniques

D.2.2.1 Top Down Design Methodology

The basic idea of top down design was presented in Section 1.0. It was pointed out that top down design showed great promise but also entailed some risk since significant changes in management practice are required. This section discusses several areas of research which are attempting to develop guidelines and tools which may reduce the risk of implementing such a management strategy.

The essence of top down programming is the division of a large program into a number of smaller subprograms (these subprograms may be subroutines, procedures, blocks, macros, etc., depending on the programming language being used). Dijkstra suggests that there are at least four ways of conceiving subprograms.

(1) Standard routines to be used as needed.

(2) Objects to be conceived by the user to reflect his analysis.

(3) A device for the reduction of program length.

(4) A means for rebuilding a given machine (computer) into a mo. suitable one.

Harlan Mills discusses the distinction between (1) and (2) above [Reference 8]. "First, we make a distinction between subprograms which are created for structuring the system and subprograms which carry out common low level processing functions in many places of the system. The latter set of subprograms we isolate first, and append to the programming language itself, just as sine or exponential routines are regarded as part of PL/I or FORTRAN. These subprograms are documented and considered as part of the language description in which the programmers write the programming system".

Mills also discusses the importance of the subprogram as a means of reducing program length. "Imagine a one hundred page PL/I program written in "GO TO" free code. Although it is highly structured, such a program is still not very readable. The extent of a major DO loop may be 50 to 60 pages, or an IF-THEN-ELSE statement may take up 10 or 15 pages. There is simply more than the eye can comfortably take in or the mind retain." Mill's solution is to impose a discipline on the top down process such that each program segment is no larger than can be contained on a single page of computer printout.

Dijkstra suggests that it is useful to view the subroutine as a means of rebuilding a given machine (computer) into a more suitable one. Starting at the top with the main program, Dijkstra chooses to view it as an entity in itself independent of the lower level subprograms. He views the main program as being executed by its own dedicated machine equipped with an adequate instruction repertoire (i.e., each of the subprogram calls are available on this hypothetical machine as primitive instructions). In

actual practice, of course, this machine will not exist (Dijkstra calls this a virtual machine). The remainder of the programming task Dijkstra sees as programming the simulation of this "virtual" machine. The process is continued in a top down manner resulting in a program arranged in "layers" or "levels". Each program level is to be understood all by itself, under the assumption of a suitable machine to execute it, while the function of each level is to simulate the machine that is assumed to be available on the level immediately above it. "The fact that a level contains "a bunch of programs" to be executed by some conceptual machine stresses the fact that the programs of this "bunch" are invited to share the same primitives."

Dijkstra further elaborated on this process in Reference 9 when he presented a concrete example of his concept of "design by levels of abstraction". The example was a multiprogramming operating system for a university computer center (the "THE Operating System".) Dijkstra's team conceived the system design as existing in several levels each of which could be understood by itself as an entity. The lower level mechanized some very detailed, difficult and machine dependent tasks (e.g., the real time clock and the interrupt structure). Above this level, however, these difficult concepts had in essence disappeared. The very primitive operations involved had been replaced by primitives on a "higher level of abstraction". In this manner, the designers of the higher levels were freed from concern with lower level details which were irrelevant to the higher level designs.

Barbara Liskow presented a design methodology which is based on structured programming in general and on "levels of abstraction" in particular [Reference 10]. An abstraction is considered as expressing "what is being done without specifying how it is done." A level is defined not only by the abstraction which it supports but also by the resources it uses to realize the abstraction. Each level has resources (e.g., I/O devices, data) which it owns exclusively and which other levels are not permitted to access. Each level is composed of a group of related functions of which there are two kinds; internal and external. Internal functions are used only within the level and cannot be referenced from other levels of abstraction. External functions may be referenced (called) only by higher level functions. Lower levels are not aware of the existence of higher levels and therefore may not refer to them in any way.

Thus, the programmer is encouraged to define subprograms for a variety of purposes (1) to "structure" his program, (2) to enhance clarity by reducing the length of other subprograms, (3) to rebuild his programming language into a language more suitable to his immediate needs and (4) to mechanize the internal and external functions of "levels of abstraction". This process is more art than science and as Mills puts it, "the programmer must use a sense of proportion and importance in identifying what is forest and what are the trees." Liskow presents a number of guidelines which may be useful in placing this rather nebulous process on a firmer basis [Reference 10].

C.A.R. Hoare presents a theory of data structuring on the premise that "It is necessary to introduce some convenient notation for expressing the abstractions involved" [Reference 1]. An algorithmic language is proposed whose purpose is to assist in the design, development and documentation of a program. This language is distinct from the programming language because "Some of the operations, although very helpful in the design of abstract programs may be very inefficient when mechanized directly on today's computers. An essential part of the program design process is to eliminate such operations in the transition from an abstract to a concrete program. The challenge of designing computers which can efficiently implement ever larger subsets of such a language may of course, be taken up in the future".

The idea of a design language separate and distinct from a programming language is certainly not new. For example, the language of ordinary algebra served this purpose for the FORTRAN programming language. The important point here is that an abstract problem solution should be developed using powerful data structures and operators pertinent to the problem being solved. The mind should not be constrained by the limitations (often severe) of the programming language being used. After the very difficult problems of what a program segment is to accomplish have been solved, most good programmers are quite adept at contriving an efficient method of implementation.

D.2.2.2 Chief Programmer Teams

The IBM Chief Programmer Team experiment is documented in References 11 and 12 and the material in this section relies heavily on these references. The experiment was the outgrowth of the work of Harlan Mills who has studied the conventional large, undifferentiated and relatively inexperienced team approach to programming projects and suggested that it might be replaced by a smaller, functionally specialized, and highly skilled team [Reference 13]. The proposed organization is compared with a surgical team in which the chief programmer is analogous to the chief surgeon and is supported by a team of specialists (as in a surgical team) whose members assist the chief rather than write parts of the program independently.

Permanent members of a Chief Programmer Team are the Chief Programmer, the Backup Programmer and a Programming Librarian. The Chief Programmer is a senior level programmer who is responsible for the detailed development of a programming system. The Backup Programmer is also a senior level programmer and works closely with the Chief Programmer to design and produce the system's key elements. The Backup Programmer has prime responsibility for system testing and also assumes the responsibility of the Chief Programmer should he leave the project. The Programming Librarian is responsible for maintenance and operation of the Program Production Library used to keep all system programs and data both internally in machine readable form and externally in well organized, highly readable form. The legal, financial, administrative and reporting requirements are coordinated by a Project Manager assigned to the team. Also, a System Analyst is assigned to the team to assist as needed. Thereafter, the team is augmented by additional programmers who produce the remainder of the code under the close supervision of the Chief and Backup Programmer.

The Chief Programmer Team is intended to solve two problems which are felt to be responsible for the relatively low productivity of current programming projects.

(1) Production projects are usually staffed by relatively inexperienced programmers at the working level and by more experienced programmers at the higher management levels. This results in several problems. First, the inexperience at the working level results in less than optimal design code and test. Second, the experienced programmers who have the insight and knowledge to correct this situation are in higher management positions where the administrative workload prevents them from effectively or economically performing any of the detailed work of programming.

The Chief Programmer Team attempts to reintroduce the highly skilled programmer into the detailed production process but free him from both the details of programming trivia and the administrative workload.

(2) In addition to normal programming activities such as design code and test, a programmer normally spends a great deal of time with what are essentially purely clerical duties. For example, he must maintain his decks and listings, punch his own corrections, setup his runs and write status reports.

The Chief Programmer Team attempts to free the programmer from all these clerical duties through the facilities of the Program Production Library and the Programming Librarian. The Program Production Library includes both machine and office procedures for performing the clerical duties of a programming project.

The Program Production Library (PPL) comprises four parts; the machine-readable internal library is a group of sub-libraries, each of which contain current project programming data. These data may be source code, relocatable modules, linkage-editing statements, object modules, job control statements, or test information. The status of the internal library is reflected in the human-readable external library binders that contain current listings of all library members and archives consisting of recently superseded listings. The machine procedures consist of standard computer steps for such procedures as the following:

* Updating libraries
* Retrieving modules for compilations and storing results
* Linkage editing of jobs and test runs
* Backing up and restoring libraries
* Producing library status listings

Office procedures are clerical rules used by librarians to perform the following duties:

* Accepting directions marked in the external library
* Using machine procedures
* Filing updated status listings in the external library
* Filing and replacing pages in the archives

A programmer using the PPL works only with the external library. Using standard conventions, he enters directly into the external library binders the changes to be made or work to be done. He then gives these changes to the librarian. Later he receives the updated external library binders, which reflect the new status of the internal library. The external library is always current and is organized to facilitate use by programmers. A chronological history of recent runs contained in the archive binders is retained to assist in disaster recovery.

The programmers are thus freed from handling decks, filing listings, keypunching, and spending unnecessary time in the machine area.

By combining standard machine procedures, standard office procedures, and project libraries, the trained librarians provide a versatile programming service that allows a team to make more effective use of its time.

The PPL also assists in improving productivity and quality by providing visibility of the work, thereby allowing team members to be aware of the status of modules that they are integrating. Such visibility also permits members to be certain of interface requirements. The internal working language of a team are the code and statements in the libraries, rather than a separate set of documents that lag behind actual status. Programmers read each other's code in order to communicate definitions, interfaces, and details of operation. Only when a question arises that cannot be resolved by reading code is it necessary to consult another programmer directly.

IBM selected the New York Times Information Bank as a project suitable for testing the validity of the Chief Programmer Team principles. This is an on-line system intended to replace the clipping file (morgue) used by the Times to provide background information for articles being written. The user views article abstracts selected by index terms and documents parameters (e.g., date, section of the paper) until he has identified those articles most relevant to his immediate needs. The user may then view the entire text of the articles selected or request that a hard copy be made. The heart of the system is the conversational subsystem and its associated data base consisting of indexing data, abstracts and complete articles. The full text of all articles is stored on microfiche and made available to the system through four TV cameras contained in a microfiche retrieval device called the RISAR that was developed by FOTO-MEM. The rest of the data base is stored on disk. Other major system components are the Data Entry Edit Subsystem, the File Maintenance Subsystem and five supporting subsystems.

Table 3 summarizes the software development tasks performed by the various members of the Chief Programmer Team assigned to the project. The numbers indicate the approximate sequence in which these tasks were performed by the various personnel. The order in which the tasks were performed was influenced by the desire to achieve a "running system" at an early date and also to achieve sufficient capability to begin building files at an early date. Otherwise, a top down approach was generally employed. As can be seen, the integration and test functions were integral parts of the development process.

Table 4 shows the staffing levels during the project. It is interesting to note the large amount of time charged by the senior personnel relative to the more junior team members (one of the goals of the Chief Programmer Team approach). This is certainly not typical of software development projects in general but some question remains as to whether this is characteristic of the Chief Programmer Team approach or whether it is due to some special characteristics of the particular project. Also of interest is the relatively short time spent on the project by most of the support personnel. This would seem to indicate that a highly flexible staffing policy may be necessary to make the Chief Programmer Team function well. Once again, management would seem to be the determining factor in the success or failure of the Chief Programmer Team (to a much greater extent than in the traditional software development process).

IBM has supplied several measures of programmer productivity and system quality achieved. Table 5 summarizes the results of acceptance testing and early operational experience with regard to errors encountered. Table 6 summarizes the programmer productivity achieved in terms of source lines of code produced per programmer day.

TABLE 3

IBM CHIEF PROGRAMMER TEAM EXPERIMENT-TASKS PERFORMED

TASKS IN APPROXIMATE CHRONOLOGICAL SEQUENCE	CHIEF PROGRAMMER	BACKUP PROGRAMMER	SYSTEM ANALYST	PROGRAMMER #1	PROGRAMMER #2	PROGRAMMER #3	PROGRAMMER #4	PROGRAMMER #5
Prepare Detailed Functional Specifications	1		1					
Develop Program Production Library (PPL) Procedures		1						
Define System Externals ('Messages, Communications Log, Statistics Reports)			2					
Design Various Subsystems and their Interfaces	2	2						
Design File Maintenance Subsystem	3	3						
Prepare Test Plan for File Maintenance Subsystem		4						
Design Preliminary On-Line System (Some Functions of Data Entry Subsystem and Conversational Subsystem)	4		3					
Develop Syntax Directed Editor for Data Entry Subsystem				1				
Program (1) Authorization File Subsystem (2) Message File Subsystem (3) Log/Statistics File Processing Subsystem (4) Deferred Print Subsystem (Hard Copy)			4					
Program Statistics Reporting Subsystem				1				
Design and Program Conversational Subsystem	5	5			1	1		
Program Terminal Handling Package							1	
Prepare Test Plans for Functional and Performance Testing		6						

NOTE: Numbers indicate the approximate sequence in which tasks were performed by the various personnel.

TABLE 4

IBM CHIEF PROGRAMMER TEAM EXPERIMENT

Staff Time (Man Months)

Work Type	Chief	Backup	Analyst	1	2	3	4	5	Technician	Manager	Sec'y	Total
Requirements Analysis	2.5	1.0	8.0	0.5	-	-	-	-	-	-	-	12.0
System Design	4.0	4.0	4.5	1.0	-	-	-	-	-	-	-	13.5
Unit design, programming, debugging and testing	12.0	14.0	10.0	13.0	4.5	2.8	3.7	4.5	-	-	-	64.5
Documentation	2.0	2.0	4.5	1.5	0.2	0.2	0.3	0.3	-	-	-	11.0
Secretarial	-	-	-	-	-	-	-	-	-	-	7.0	7.0
Librarian	-	-	-	-	-	-	-	-	5.5	-	2.0	7.5
Manager	3.5	2.0	-	-	-	-	-	-	-	11.0	-	16.5
TOTAL	24.0	23.0	27.0	16.0	4.7	3.0	4.0	4.8	5.5	11.0	9.0	132.0

TABLE 5

IBM CHIEF PROGRAMMER TEAM EXPERIMENT

ERRORS IDENTIFIED DURING ACCEPTANCE TESTING

SUBSYSTEM	SOURCE LINES	Error Type* INCORRECT FUNCTION	OMITTED FUNCTION	MISINTERPRETED FUNCTION	TOTAL	
File Maintenance	12,029	0	0	0	0	
Conversational	38,990	9	8	3	20	
Data Entry Edit	13,421	0	0	1	1	
Other	18,884	0	0	0	0	
TOTAL	83,324	9	8	4	21	

ERRORS IDENTIFIED DURING OPERATION

SUBSYSTEM	SOURCE LINES	Error Type* INCORRECT FUNCTION	OMITTED FUNCTION	MISINTERPRETED FUNCTION	TOTAL	
File Maintenance	12,029	1	0	1	2	
Conversational	38,990	4	3	0	7	
Data Entry Edit	13,421	8	5	3	16	
Other	18,884	0	0	0	0	
TOTAL	83,324	13	8	4	25	

* "INCORRECT FUNCTION" - refers to code which operated improperly.

"OMITTED FUNCTION" - refers to specifications not implemented.

"MISINTERPRETED FUNCTION" - refers to code which did not perform precisely the functions specified.

TABLE 6

IBM CHIEF PROGRAMMER TEAM EXPERIMENT - PROGRAMMER PRODUCTIVITY

Organization	Source lines per programmer day
Unit design, programming, debugging, and testing	65
All professional	47
With librarian support	43
Entire team	35

*The first row includes work done on unit design, coding, debugging, and acceptance testing. The second row summarizes professional work, which includes system design and documentation, but not librarian support. The third row includes all programming and librarian support. The last row presents the productivity of the entire team on the completed system (excluding requirements analysis).

D.2.2.2 Computer Program Management Technique (CPMT)

CPMT is a systematic discipline for managing the development, verification and documentation of scientific and engineering computer programs. It incorporates many of the design, coding and testing concepts which have been developed through structured programming research. It also incorporates much of the Chief Programmer Team management philosophy although specific concepts have been modified somewhat to make them more compatible with the aerospace scientific-engineering environment.

CPMT defines personnel assignments in terms of functions. Thus, for a small program, several functions may be performed by the same person whereas for large programs a single function may require several people. The following functions are identified:

Requestor - defines the program requirement in order to solve some specific problem.

Study Manager - supervises the program development and ensures the implementation of CPMT procedures.

Principal Investigator - responsible for the program meeting the technical requirements.

Chief Programmer - responsible for the structural design of the program and the coding of high level or control components.

Engineer/Programmer - responsible for coding, documentation and testing.

Librarian - maintains the program workbook, coordinates all documentation and compiles project statistics (i.e., actual versus projected results).

CPMT identifies five phases of program development:

(1) Planning - A study plan is prepared defining the proposed method of solution, potential problems, and schedule and cost estimates.

(2) Design - The structural design of the program is developed. Test plans and acceptance criteria are also defined during this phase.

(3) Development - The coding, subprogram documentation and unit testing of the program is performed in this phase.

(4) Program Test and Verification - The complete program is tested during the phase and final documentation is prepared.

(5) Release - A Program Manual and a CPMT Study Report are released. A document specifying baseline test cases is prepared. A configuration control manager for the program is assigned and the program is released to the customer.

CPMT documentation is designed for ease of preparation. Skeleton documents are prepared as early as practical and details are added when they become available. Preparation of final reports is intended to be mainly an exercise of putting the finishing touches on working documentation which already exists. The clerical aspects of documentation are handled by the librarian so the programmer is freed to a large extent from document editing and rewrites.

Management reviews are held throughout the program development process with formal reviews scheduled at the conclusion of each of the five phases discussed above. Figure 5 indicates the relationships between personnel functions, program phases, formal reviews and the more important documentation requirements. CPMT procedures are described in detail in a proprietary document of McDonnell Douglas Corporation [Reference 14].

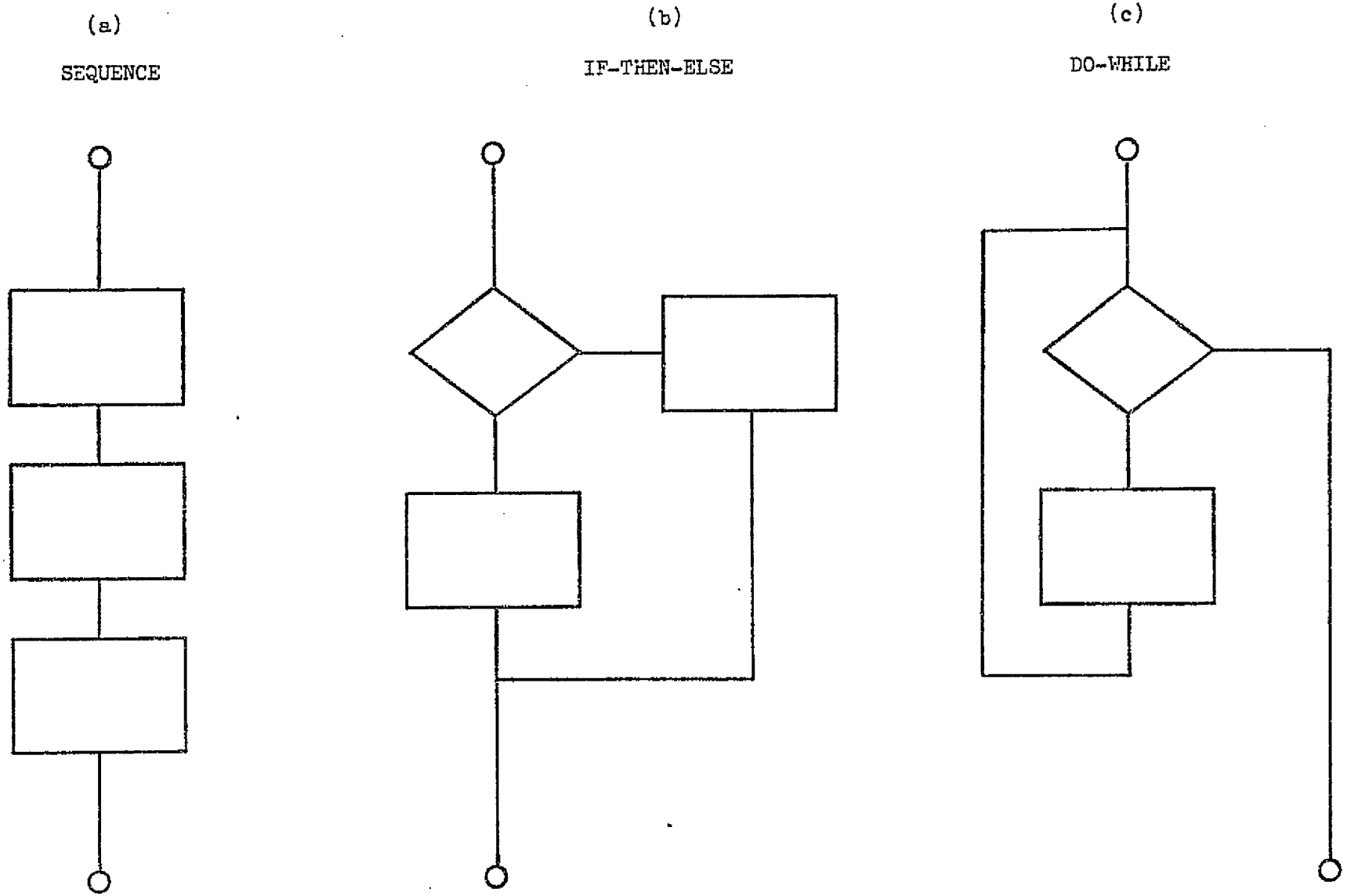
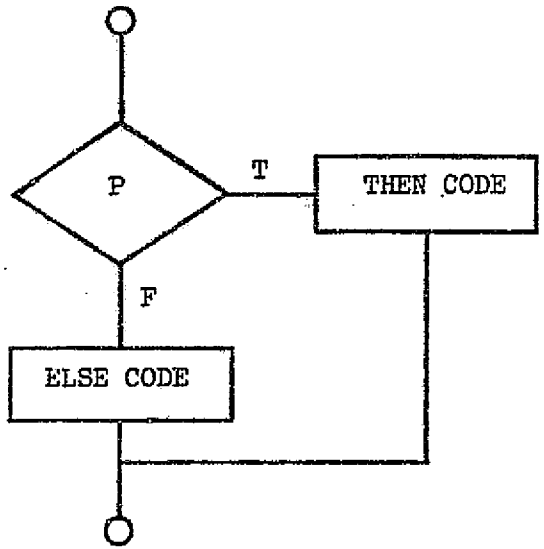
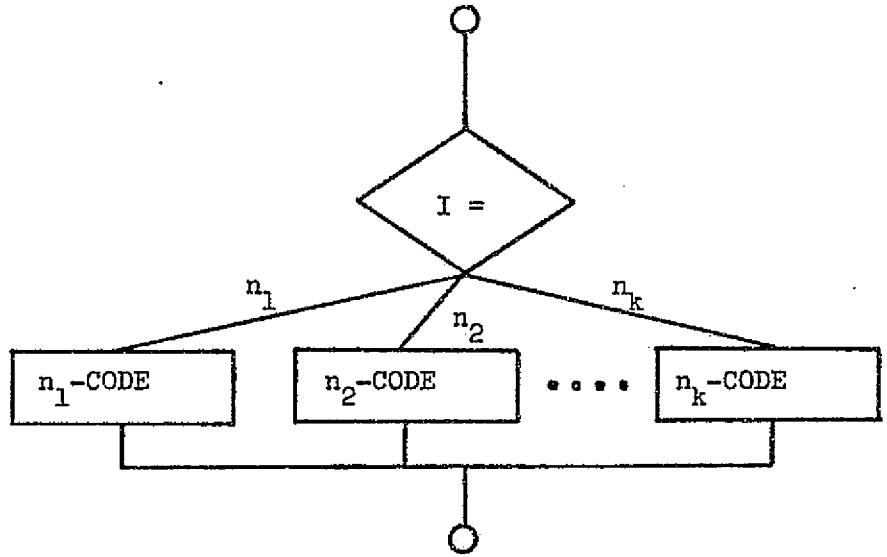


FIGURE 1

THREE BASIC CONTROL STRUCTURES

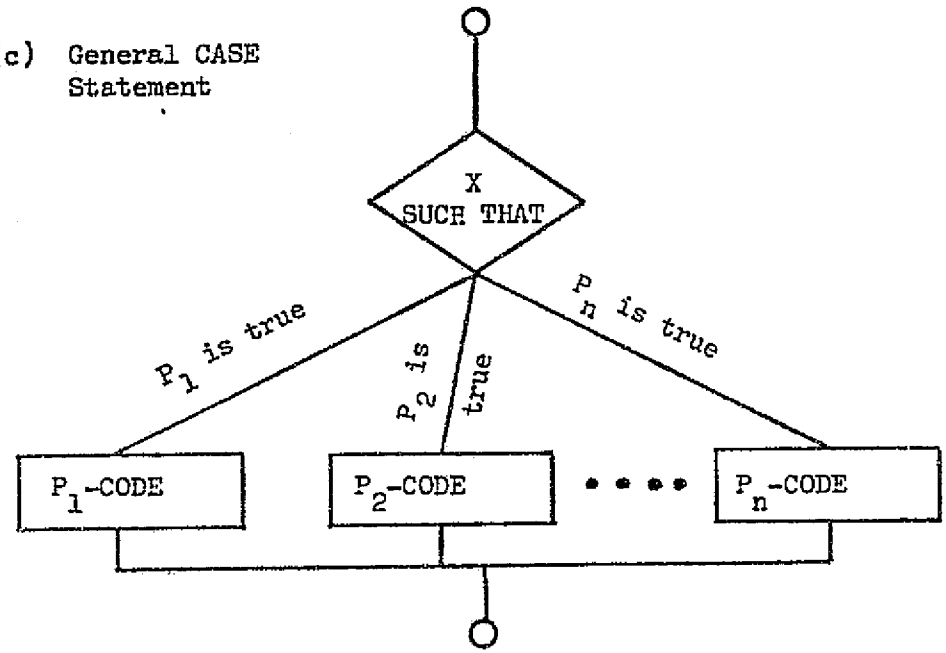


(a) IF-THEN-ELSE



(b) Simple CASE Statement

(c) General CASE Statement



Example: P_1 is the statement " X = 10 "

P_2 is the statement " X greater than 10 and X less than 100 "

P_n is the statement " X = 1000 "

FIGURE 2

IF-THEN-ELSE AND MORE GENERAL DECISION CONSTRUCTS

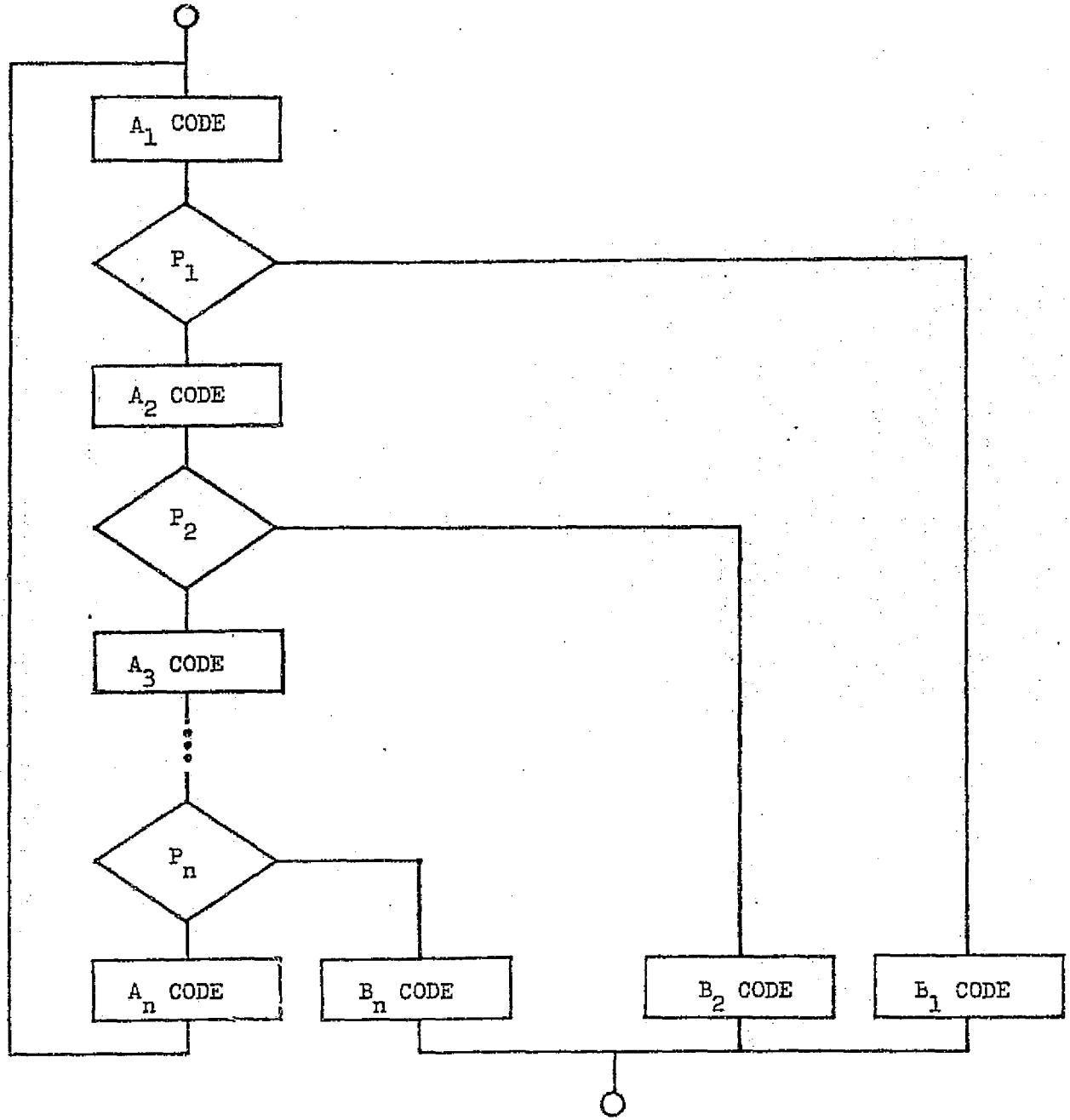
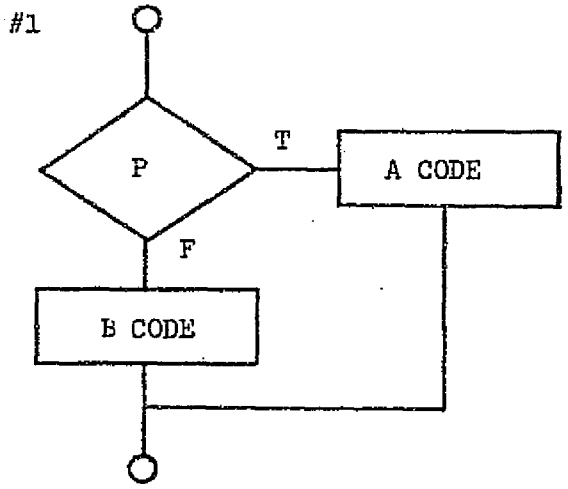
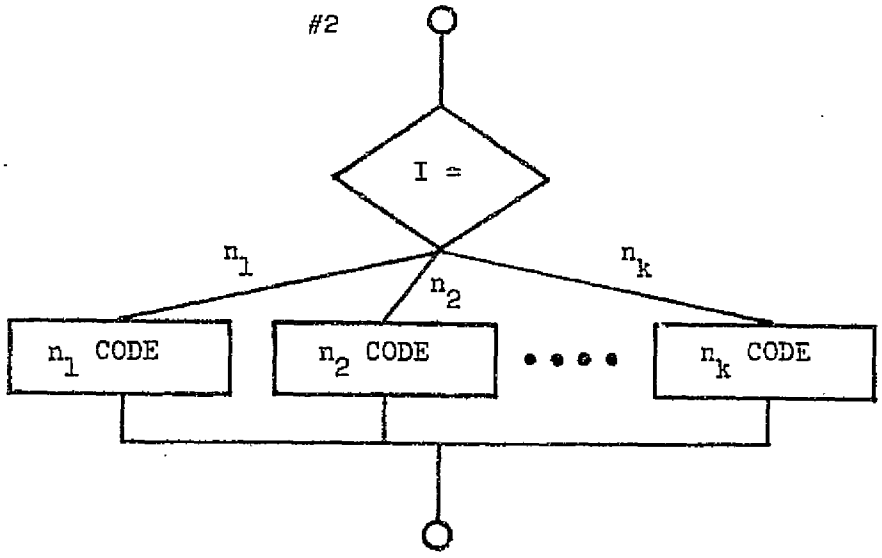


FIGURE 3

A GENERAL LOOP CONSTRUCT





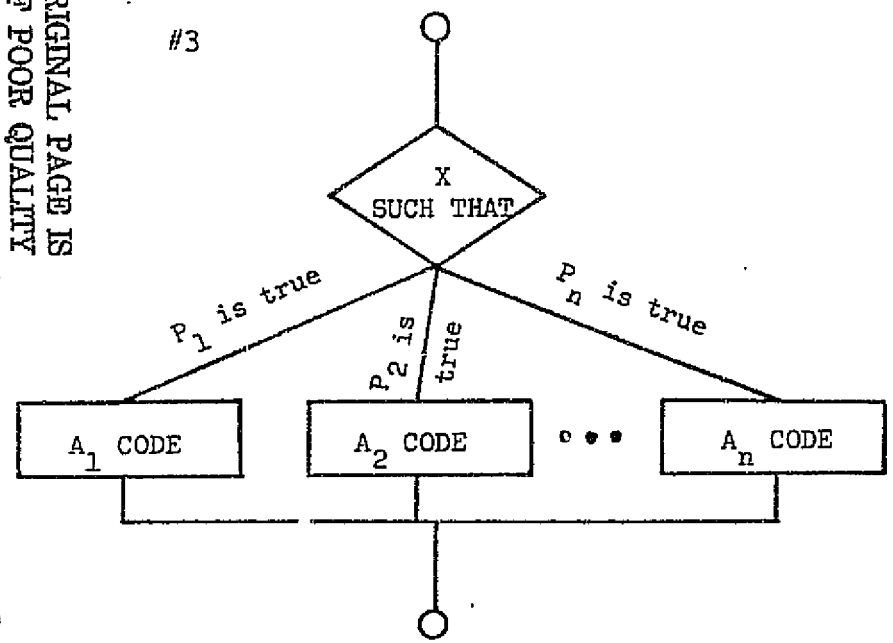

```
IF (P)                  IF (.NOT.(P)) GO TO α
  THEN                    A CODE
    A CODE                GO TO β
  ELSE                  α CONTINUE
    B CODE                B CODE
                        β CONTINUE
```

```
CASE OF I               IF (I.NE.n1) GO TO α1
CASE n1                   n1 CODE
  n1 CODE                 GO TO β
CASE n2                 α1 CONTINUE
  n2 CODE                 IF (I.NE.n2) GO TO α2
  .                         n2 CODE
  .                         GO TO β
  .                     α2  CONTINUE
CASE nk                   .
  nk CODE                 .
                          .
                        αk-1 CONTINUE
                          IF (I.NE.nk) GO TO β
                            nk CODE
                        β   CONTINUE
```

FIGURE 4

SOME REPRESENTATIVE CONSTRUCTS FOR STRUCTURED PROGRAMMING IN FORTRAN

#3




```
CASE OF X
CASE (P1)
  A1 CODE
CASE (P2)
  A2 CODE
   .
   .
   .
CASE (Pn)
  An CODE
```

```
      IF (.NOT.(P1)) GO TO α1
        A1 CODE
        GO TO αn
α1    CONTINUE
      IF (.NOT.(P2)) GO TO α2
        A2 CODE
        GO TO αn
α2    CONTINUE
        .
        .
        .
αn-1  CONTINUE
      IF (.NOT.(Pn)) GO TO αn
        An CODE
αn    CONTINUE
```

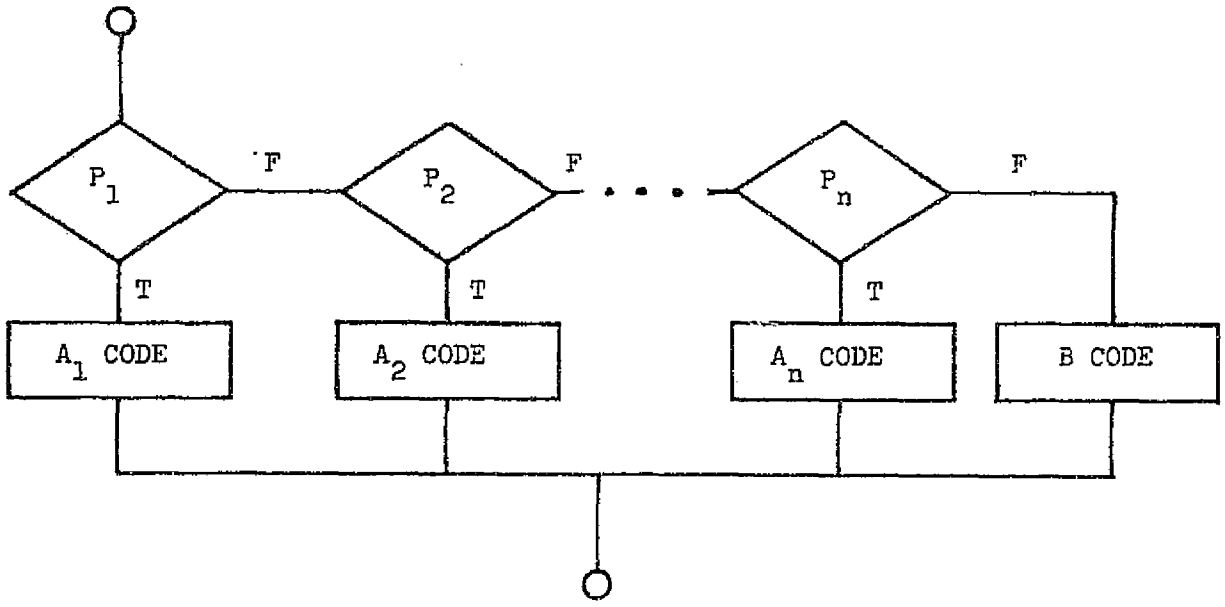

Note: The CASE statement involves the choice of a particular path based on the value of a logical expression (P), the simplest form of which is:

X R Y

where X and Y are variables (or expressions) and R is a relation.

In the simplest form of CASE statement (#2) the first two of these quantities are assumed to be the same for all paths (e.g., X is the variable I shown in the Figure, R is the relation "=" and a value of the variable Y is associated with each path). A more general form of the CASE statement (#3) allows both R and Y (but not X) to vary from path to path. (E.g., this form allows taking the first path if X is less than 6, the second path if X is greater than 10 and the third path if X is equal to 7.) The most general form of the CASE statement (#4) allows all three quantities (X, R and Y) to vary from path to path.

FIGURE 4 (cont'd)




```
SELECT BLOCK
(P1)
    A1 CODE
(P2)
    A2 CODE
  :
(Pn)
    An CODE
(.NOT.(Pn))
    B CODE
ENDBLOCK
```

or

```
IF (P1)
  THEN
    A1 CODE
ORIF (P2)
  THEN
    A2 CODE
  :
ORIF (Pn)
  THEN
    An CODE
ELSE
    B CODE
ENDIF
```

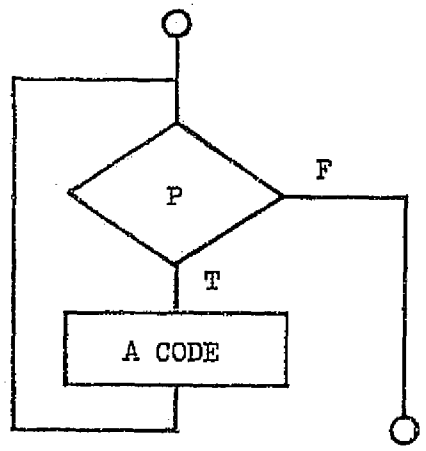
or

```
SELECT ONE
CASE (P1)
  A1 CODE
CASE (P2)
  A2 CODE
  :
CASE (Pn)
  An CODE
CASE (.NOT.(Pn))
  B CODE
```

```
      IF (.NOT.(P1)) GO TO α1
      A1 CODE
      GO TO β
α1    CONTINUE
      IF (.NOT.(P2)) GO TO α2
      A2 CODE
      GO TO β
α2    CONTINUE
      :
αn-1  CONTINUE
      IF (.NOT.(Pn)) GO TO αn
      An CODE
      GO TO β
αn    CONTINUE
      B CODE
β     CONTINUE
```

FIGURE 4 (cont'd)

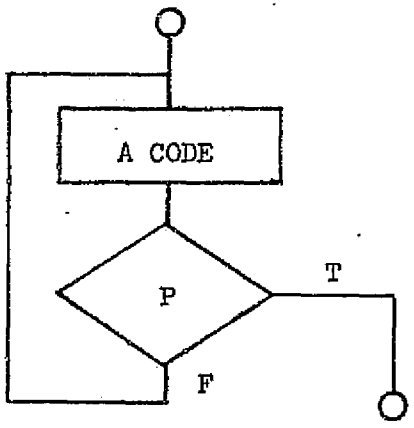
#5




```
DOWHILE (P)
  A CODE
ENDDO
```

```
α  CONTINUE
   IF (.NOT.(P)) GO TO β
   A CODE
   GO TO α
β  CONTINUE
```

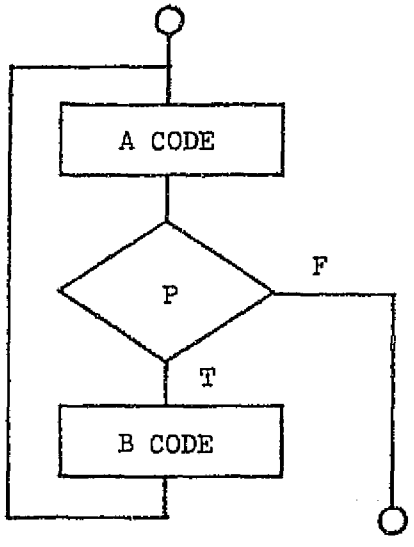
#6




```
DO UNTIL (P)
  A CODE
ENDDO
```

```
α  CONTINUE
   A CODE
   IF (.NOT.(P)) GO TO α
   CONTINUE
```

#7




```
REPEAT BLOCK
  A CODE
EXIT BLOCK (P)
  B CODE
ENDBLOCK
```

or

```
DOWHILE (P)
  B CODE
BEGINWHILE
  A CODE
ENDDO
```

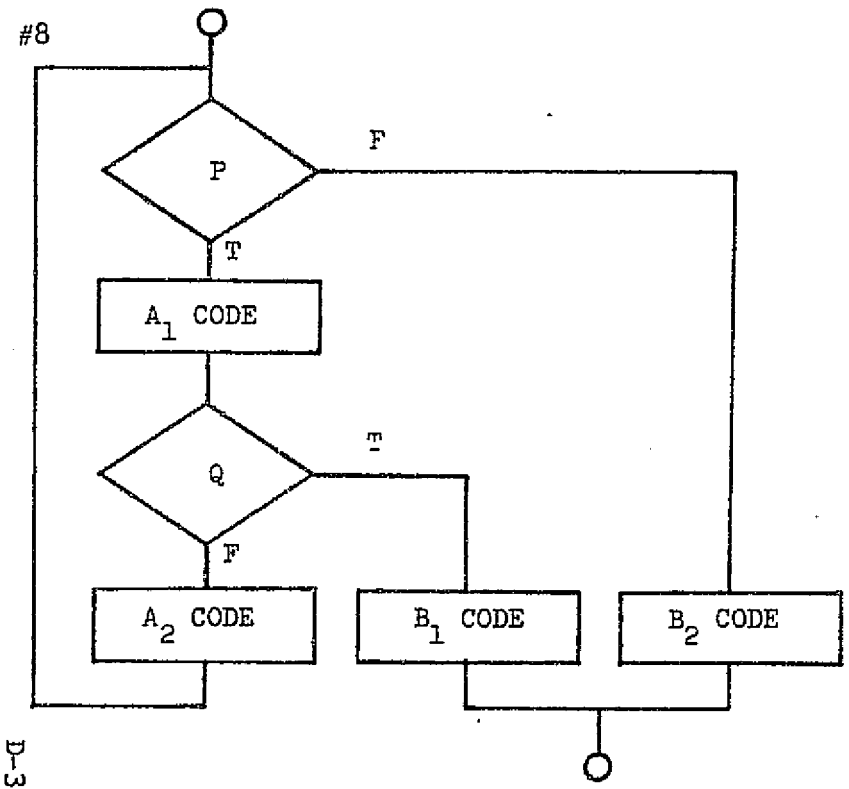
or

```
BEGINLOOP
  A CODE
EXIT IF (P)
  B CODE
ENDLOOP
```

```
α  CONTINUE
   A CODE
   IF (.NOT.(P)) GO TO β
   B CODE
   GO TO α
β  CONTINUE
```

FIGURE 4 (cont'd)

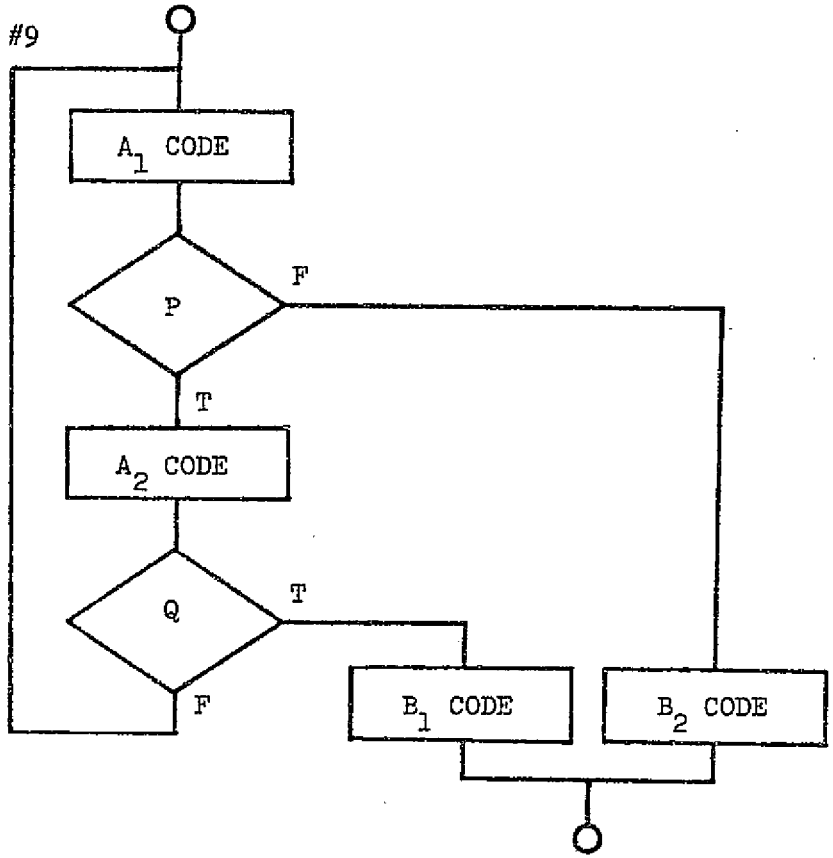
#8




```
STARTSEARCH
WHILE (P)
  A1 CODE
EXITIF (Q)
  B1 CODE
ORELSE
  A2 CODE
ENDLOOP
  B2 CODE
ENDSEARCH
```

```
α   CONTINUE
    IF (.NOT.(P)) GO TO α1
    A1 CODE
    IF (Q) GO TO α2
    A2 CODE
    GO TO α
α1  CONTINUE
    B2 CODE
    GO TO β
α2  CONTINUE
    B1 CODE
β   CONTINUE
```

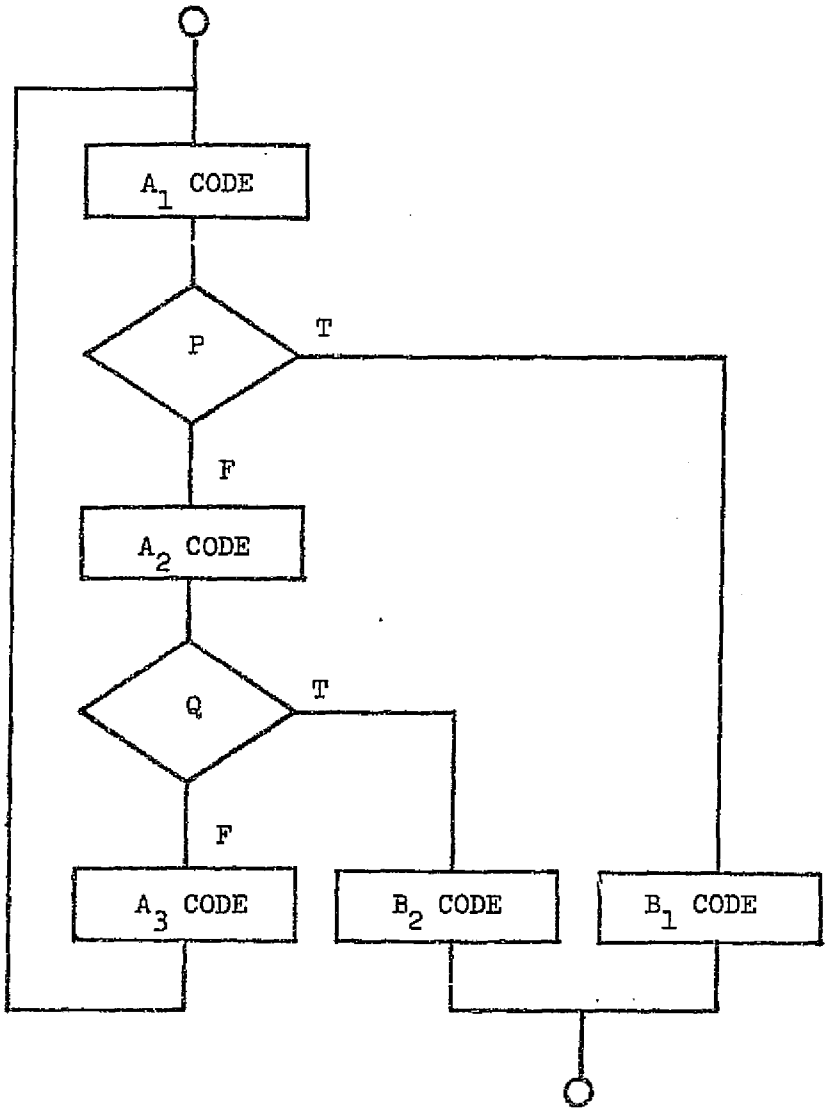
#9




```
STARTSEARCH
UNTIL (P)
  A1 CODE
EXITIF (Q)
  B1 CODE
ORELSE
  A2 CODE
ENDLOOP
  B2 CODE
ENDSEARCH
```

```
α   CONTINUE
    A1 CODE
    IF (.NOT.(P)) GO TO α1
    A2 CODE
    IF (.NOT.(Q)) GO TO α
    B1 CODE
    GO TO β
α1  CONTINUE
    B2 CODE
β   CONTINUE
```

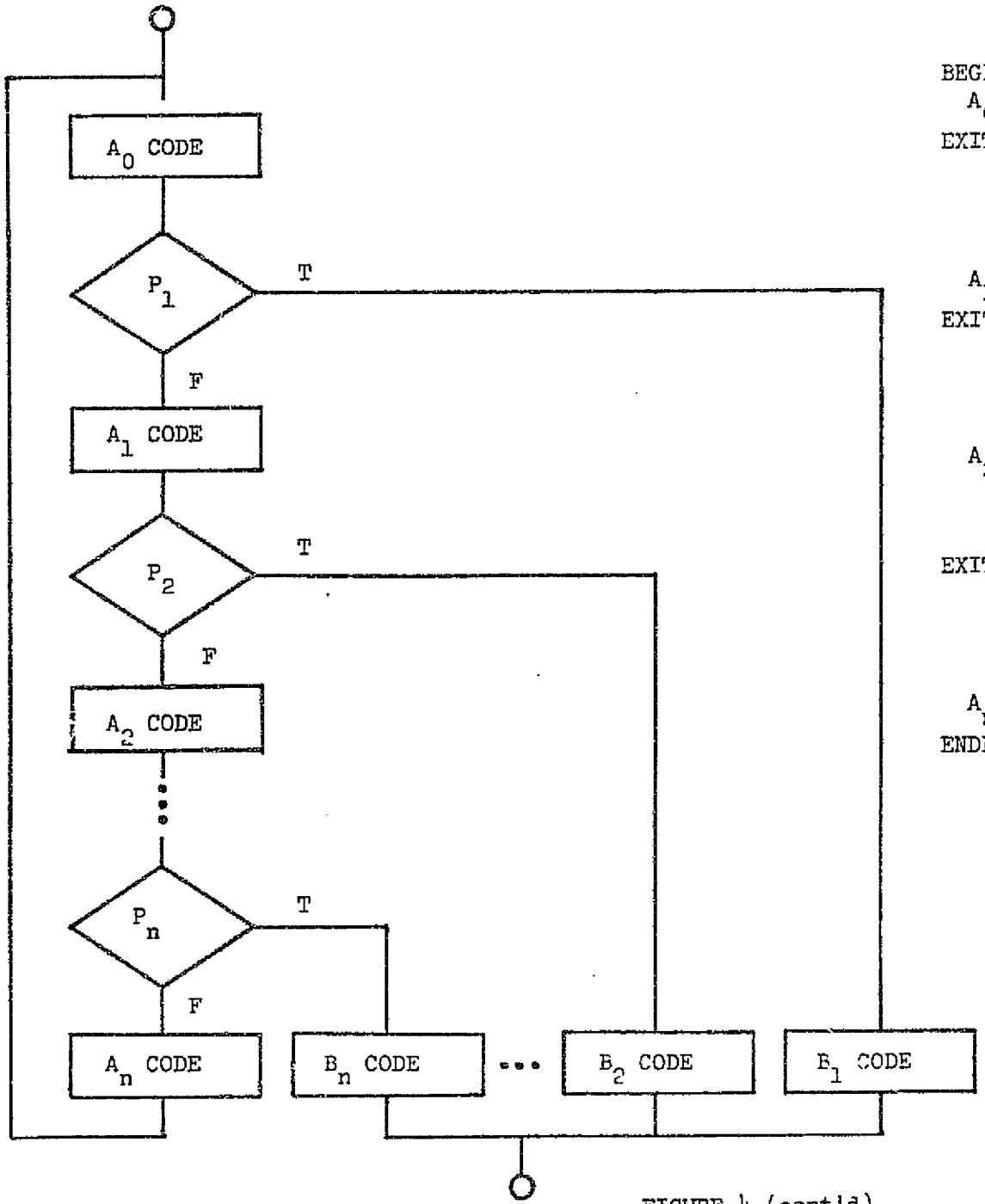
FIGURE 4 (cont'd)




```
BEGINLOOP
  A1 CODE
EXITIF (Q)
    THEN
      B1 CODE
    ENDIFEXIT
  A2 CODE
EXITIF (Q)
    THEN
      B2 CODE
    ENDIFEXIT
  A3  CODE
ENDLOOP
```

```
α   CONTINUE
    A1 CODE
    IF (.NOT.(P)) GO TO α1
    B1 CODE
    GO TO β
α1  CONTINUE
    A2 CODE
    IF (.NOT.(Q)) GO TO α2
    B2 CODE
    GO TO β
α2  CONTINUE
    A3 CODE
    GO TO α
β   CONTINUE
```

FIGURE 4 (cont'd)




```
BEGINLOOP                α  CONTINUE
  A0 CODE                   A0 CODE
EXITIF (P1)                 IF (.NOT.(P1)) GO TO α1
     THEN                   B1 CODE
        B1 CODE             GO TO β
     ENDIFEXIT           α1 CONTINUE
  A1 CODE                   A1 CODE
EXITIF (P2)                 IF (.NOT.(P2)) GO TO α2
     THEN                   B2 CODE
        B2 CODE             GO TO β
     ENDIFEXIT           α2 CONTINUE
  A2 CODE                   A2 CODE
     .                          .
     .                          .
     .                          .
EXITIF (Pn)                 IF (.NOT.(Pn)) GO TO αn
     THEN                   Bn CODE
        Bn CODE             GO TO β
     ENDIFEXIT           αn CONTINUE
  An CODE                   An CODE
ENDLOOP                     GO TO α
                            β  CONTINUE
```

FIGURE 4 (cont'd)

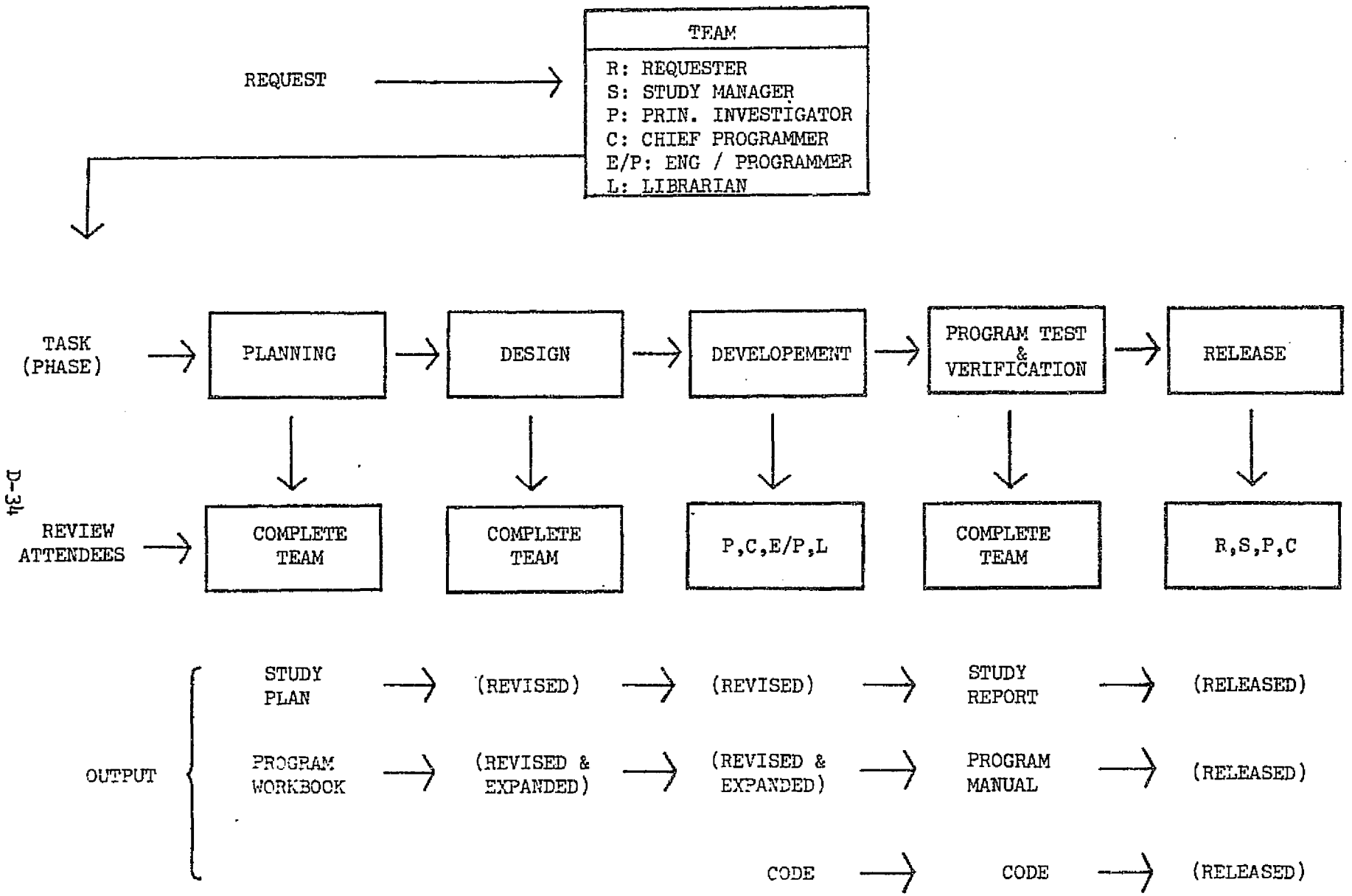


FIGURE 5

CPMT PROCEDURE

REFERENCES

1. O. J. Dahl, E. W. Dijkstra, C.A.R. Hoare. Structured Programming (Book) Academic Press 1972.

2. C.A.R. Hoare. Proof of a Program: FIND. Comm. ACM, January 1971, pp. 39-45.

3. D. Bohm, G. Jacopini. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. Comm. ACM, May 1966, pp. 366-371.

4. E. W. Dijkstra. GOTO Statement Considered Harmful. Letter to the Editor, Comm. ACM March 1968, pp. 147-148.

5. Niklaus Wirth. Systematic Programming An Introduction (Book) Prentice Hall, 1973.

6. R. Evans. Multiple Exits from a Loop Using Neither GOTO nor Labels. Short Communications - Comm. ACM Nov. 1974, page 650.

7. G. Bachmann. Multiple Exits from a Loop Without the GOTO. Comm ACM, July 1973, pages 443-444.

8. H. Mills. Structured Programming. IBM Federal Systems Division dated October 1970.

9. E. W. Dijkstra. The Structure of the THE Multiprogramming System. Comm. ACM May 1968, pp. 341-346.

10. B. Liskow. A Design Methodology for Reliable Software Systems. Proc. FJCC 1972 pages 191-199.

11. F. T. Baker. Chief Programmer Teams. IBM Systems Journal, Vol. 11, No. 1, 1972.

12. F. T. Baker. System Quality Through Structured Programming. Proc. FJCC (1972) pp. 339-343.

13. H. Mills. Chief Programmer Teams: Principles and Procedures. IBM Federal System Division No. FSC71-5108, June 1971.

14. Computer Program Management Technique (CPMT), McDonnell Douglas Astronautics Company - West - Manual #78.

15. P. Naur. Proof of Algorithms By General Snapshots. BIT 6 1966, pp. 310-316.

16. R. W. Floyd. Assigning Meanings to Programs. American Mathematical Society -- Mathematical Aspects of Computer Science, Vol. 19, 1967, pp. 19-32.

17. E. W. Dijkstra. A Constructive Approach to the Problem of Program Correctness. BIT Vol. 8, No. 3, 1968, pp. 174-186.

18. C.A.R. Hoare. An Axiomatic Basis for Computer Programming. Comm. ACM, October 1969, pp. 576-583.

19. Z. Manna. Properties of Programs and the First Order Predicate Calculus, J.ACM, April 1969, pp. 244-255.

20. B. Elspas, K. N. Levitt, R. J. Waldinger, A. Waksman. An Assessment of Techniques for Proving Programs Correct. ACM Computing Surveys, June 1972, pp. 97-147.

21. N. J. Nilsson. Problem Solving Methods in Artificial Intelligence. (Book) McGraw-Hill, 1971.

Appendix E

PROVING PROGRAMS CORRECT

E.1 INTRODUCTION

The first serious notion that programs could and should be proved correct is probably the pioneering work of Peter Naur (Reference 1) and Robert Floyd (Reference 2). The methods propounded were quite similar and were developed independently. Naur's method was based on what he called "general snapshots" and was a rather informal (though rigorous) conception of a proof. Floyd's approach was somewhat more formal and several concepts fundamental to the modern formal proof methods are present in his paper. In particular 1) the use of formal mathematical logic, 2) the idea of an "abstract program" and 3) the idea of an "interpretation" of an abstract program.

Program proving over the years has grown in two different directions which can probably be best described as formal and informal. Informal program proving is most often encountered in the literature generally associated with structured programming whereas formal program proving is usually encountered in the literature on artificial intelligence. The informal methods have the disadvantage that there are few underlying general principles and each problem presents a separate challenge. The advantage of the informal method is that the human "prover" may use any notation which fits the current problem and use any method of proof suitable to the particular problem at hand. The formal methods on the other hand have developed to the point where there is a quite solid mathematical basis. There are, however, two related and rather serious problems with the formal methods at the present time.

The formal methods require that either a simple language (e.g., the first order predicate calculus) or a complex language (e.g., second or higher order mathematical logic) be used for the mechanics of the proof. If the simple language is chosen there are reasonably efficient proof methods available but one encounters extreme difficulty in formulating "real" problems because certain basic mathematical concepts (most notably the equality relation) are not easily expressed in this language. If the complex language is chosen, the formulation problem is eased considerably but there are as yet no really satisfactory proof methods available. Before a really practical application of the formal proof methods can be made it is probable that greatly improved proof methods for higher order logics will be required.

E.2 INFORMAL PROOF METHODS

The literature on informal proofs of programs consists almost entirely of sample demonstrations for particular algorithms. At least two different approaches are identifiable: the constructive approach and the verification approach. In the constructive approach (Dijkstra, Reference 3) an algorithm proof is developed in a top down manner from the algorithm specifications. The steps of the algorithm proof are then converted to executable code in what is normally a relatively trivial exercise. The result is an algorithm which has been "proved correct" and then converted to executable code. It is important to note that it is the algorithm not the code that has been proved correct.

In the verification approach (Floyd, Reference 2) the executable code is assumed to exist. The algorithm proof steps referred to above either exist or must be generated. Figure E-1 illustrates the verification problem given the code in flow chart form and the algorithm proof steps in the form of statements in mathematical logic (propositions).

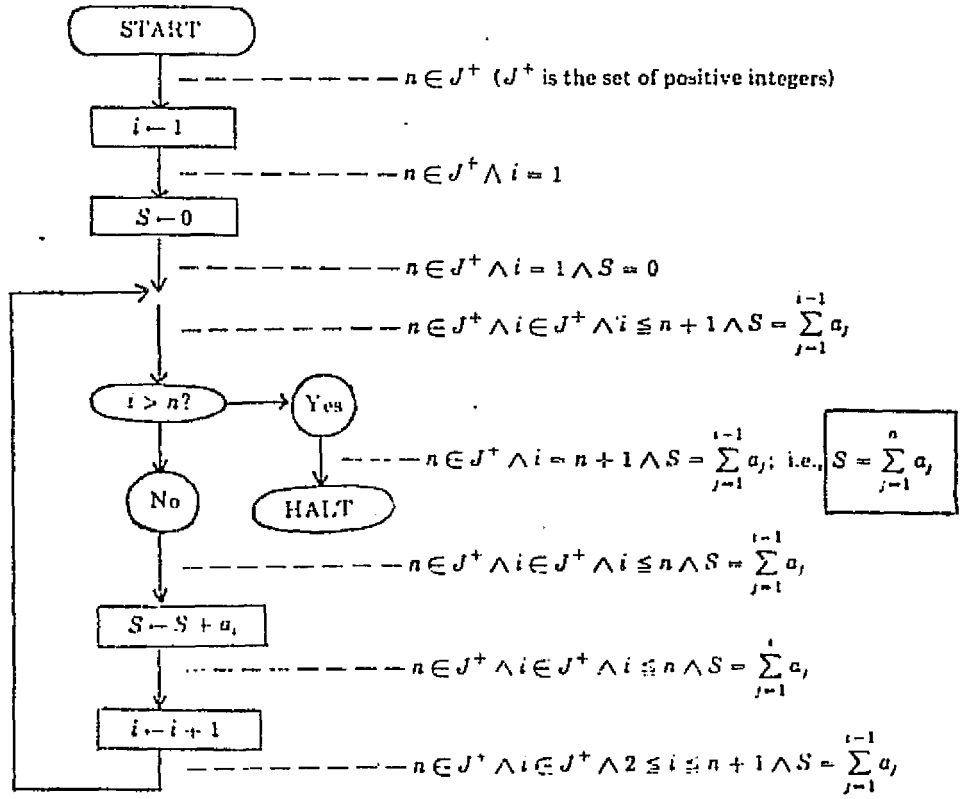


Figure E-1. Flowchart Illustrating Floyd's Method of Program Verification

The code between propositions Floyd calls "commands." On the flowchart, these commands are connected by "arrows" representing the possible passages of control between the commands. Each command (except START and HALT) has at least one "entrance" arrow (a_i) and at least one "exit" arrow (b_j). A "proposition" is associated with each of these arrows. Thus each command has one or more entrance propositions (P_i) and one or more exit propositions (Q_j). Using this terminology Floyd defines a verification as "a proof that for every command c of the flowchart, if control should enter the command by an entrance (a_i) with P_i true, then control must leave the command, if at all, by an exit (b_j) with Q_j true." The entrance and exit propositions Floyd calls the "verification conditions" for a command. These verification conditions are identical to the "algorithm proof steps" generated in the constructive approach. Thus "verification" bridges the gap between an algorithm proof and a proof of its representation in executable code.

As the reader has probably determined, proving a program correct requires two very difficult steps: 1) determining "verification conditions" which faithfully represent the desired algorithm, and 2) performing the proof required by the above definition of a verification. Unfortunately the literature offers little guidance except by specific example. One exception to this is Hoare's concept of invariants (References 4 and 5) which appears to offer solid foundations in an area where few exist.

Invariants

Hoare (Reference 4) defines an invariant as "a formula of logic which is intended to remain true throughout the execution of a program segment" (even though the values of any variable appearing in the formula may be changed by the execution). One reason that invariants are important is because they provide a very useful insight into how a loop performs a desired function. To be used in this manner, an invariant is required whose meaning is essentially a specification of what the loop is intended to accomplish. Formulation of the specifications of a program segment (e.g., a loop) in invariant form is a step which sometimes requires great ingenuity. The basic idea, however, can be illustrated by the simple example shown in Figure E-2. The key step is the expression of the program specifications in invariant form. In this simple example this is accomplished by replacing the parameter N (which is important only to a final result) by the parameter J (which has significance for all intermediate results). It should be noted that upon termination J = N and the two specifications are the same. The invariant form however, is true throughout execution of the loop especially at the points labeled ① ② ③ and ④ in Figure E-2. The non-invariant form is necessarily true only at point ④ .

Problem: Find the maximum value of an array A of dimension N. Set B equal to this value.

Program Segment Specification:

For all I such that $(1 \leq I \leq N)$ $B \geq A[I]$
For at least one I such that $(1 \leq I \leq N)$ $B = A[I]$

Program Segment Specification In Invariant Form:

For all I such that $(1 \leq I \leq J)$ $B \geq A[I]$
For at least one I such that $(1 \leq I \leq J)$ $B = A[I]$
For Loop Termination J = N

Program:

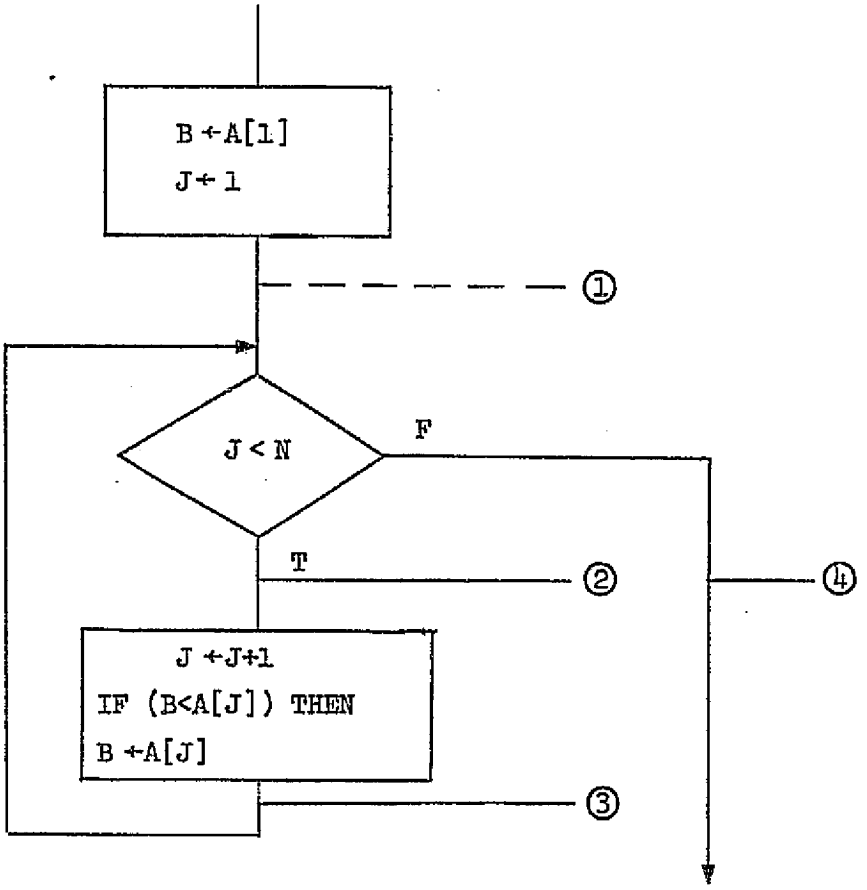


Figure E-2. Program Specification With Invariants

Once the invariant for a loop has been determined, the formal proof of loop correctness is straightforward. The idea is to prove that the invariant is true upon exit from the loop (i.e., at point ④ in Figure E-2). This is done by two steps which are:

(1) Prove the invariant is true before the loop is entered (i.e., at point ① in Figure E-2).

(2) (a) assume the exit condition is not satisfied (i.e., J<N)

(b) assume the invariant is true at point ②

(c) mentally execute the body of the loop once, i.e., (J←J+1; IF (B<A[J]) THEN B←A[J])

(d) prove that the invariant remains true. (i.e., that it is true at ③)

The above two steps and the principle of mathematical induction are sufficient to prove the desired result - namely that the invariant is true upon exit from the loop. Loop termination is proved separately and will establish that upon exit J=N which makes the invariant form of program specification identical to the original program specification.

E.3 FORMAL PROGRAM PROVING

Formal program proving attempts to overcome a very serious drawback of the informal methods - i.e., the necessity of dealing with each program on an individual basis. To do this it is necessary to abstract the concept of a program - to identify the essential "structure" of a program and to eliminate the details which are peculiar to a certain representation of that program. The result is a "program schema" or "abstract program" which is a sort of skeleton program consisting solely of assignment statements and branch statements. More specifically an abstract program consists of the following:

(1) A vector of input variables x

(2) A vector of program variables y

(3) A vector of output variables z

(4) A vector of program constants a

(5) Assignment statements of the form

$$y \leftarrow f(x, y)$$

(6) Branch statements consisting of a predicate (logical expression) $P_i(x,y)$ where either of two paths are taken depending on the truth of falsity of $P_i(x,y)$.

Actual input and output do not occur in an abstract program. Rather this is handled by assignment statements (i.e., input is accomplished by assigning a function of input variables to a vector of program variables, e.g., $y \leftarrow g(x)$. Similarly output is accomplished by assigning a function of input and program variables to a vector of output variables ($z \leftarrow h(x,y)$). Figure E-3 shows an abstract program in flowchart form.

An "interpretation" of an abstract program specifies

(1) Specific functions and predicates

(2) Specific values for all program constants

(3) The domains of the input, program and output variables. (Note in particular that values for input variables are not assigned - merely the domain - e.g., an input variable may be constrainted to be a positive integer but its value is unspecified).

Under an "interpretation" an "abstract program" becomes an actual program capable of execution once the values of the input variables are specified. Thus an "interpretation" forms the link between abstract programs and actual executable programs.

The theoretical basis for the formal approach is due to Manna (Reference 6) who showed in essence that the verification of any abstract program can be converted into the proving of a theorem (usually) in the first order predicate calculus. The development which follows is based on Reference 7 and to a limited extent assumes the reader is familiar with the predicate calculus (Reference 8, chapter 6 is a reasonably straightforward development of the predicate calculus for the reader desiring more background). Before proceeding with the formalism however, it is necessary to firm up some basic concepts.

Roughly speaking, a program may be said to be correct if its execution terminates and it yields the desired final result. However, since both termination of execution and attainment of a desired result usually depend on the input vector (x), it is necessary to introduce a predicate $\phi(x)$ representing these constraints. Likewise, it is necessary to formalize the idea of "attaining the desired final result" by introducing a predicate $\psi(x,z)$ which is true if and only if z is the desired output for valid input x. Thus we may speak of a program being correct with respect to the input predicate $\phi(x)$ and the output predicate $\psi(x,z)$.

It has been found useful to define two types of program correctness primarily because a proof of program termination is often (but not always) most easily performed separate from the proof of correctness. Thus a program is said to be correct with respect to input predicate $\phi(x)$ and output predicate $\psi(x,z)$ if it yields the correct answer (i.e., satisfies $\psi(x,z)$) and if it terminates for all valid input (i.e., input vectors x satisfying $\phi(x)$). Alternatively, a program is said to be partially correct with respect to input predicate $\phi(x)$ and output predicate $\psi(x,z)$ if it yields the correct answer when it terminates (for valid input x). A proof of termination together with a proof of partial correctness is of course equivalent to a proof of correctness.

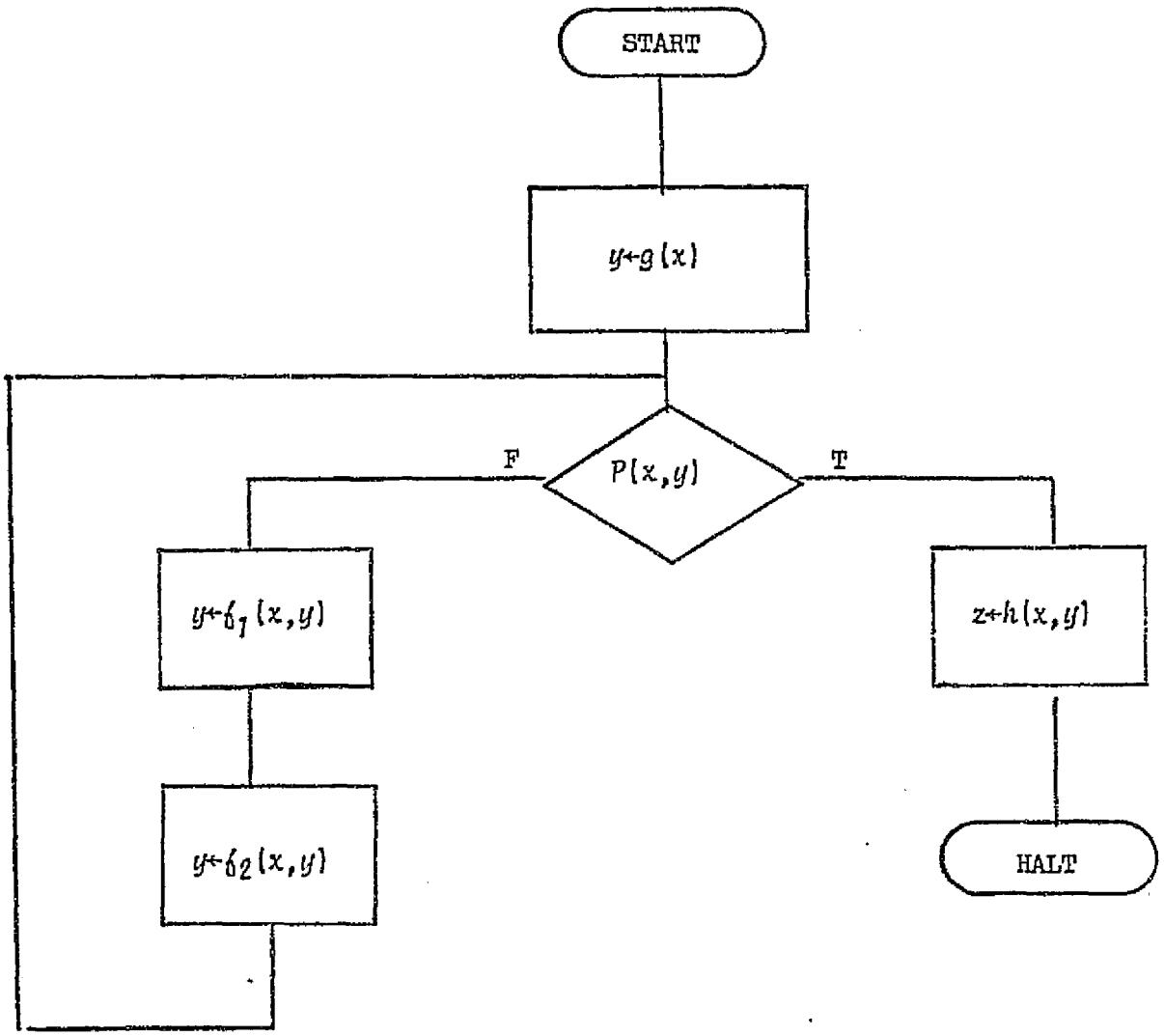


Figure E-3. An Abstract Program Flowchart

Manna's formalism assumes an abstract program to consist of a series of statements of the form

I: IF $P_i(x,y)$

THEN
$y \leftarrow f_i^1(x,y)$

GOTO I_1

ELSE
$y \leftarrow f_i^2(x,y)$

GOTO I_2

where I; I_1; I_2 are statement labels

$P_i(x,y)$ is a predicate

$f_i^1(x,y)$; $f_i^2(x,y)$ are functions

In the above standard form, any of the go to statements may be replaced by the HALT command which indicates program termination. It is a relatively straightforward exercise to convert any abstract program to this standard form.

With each statement in the above standard form, Manna associates a "well formed formula" in the predicate calculus: ("well formed formula" is essentially a statement expressed in mathematical logic which is either true or false depending on the values of the variables contained in it).

$$W_i = \forall_y \, q_i(x,y) \Rightarrow$$
$$\left\{\begin{array}{l} \text{IF } P_i(x,y) \\ \quad \text{THEN } q_{i_1}(x, f_i^1(x,y)) \\ \quad \text{ELSE } q_{i_2}(x, f_i^2(x,y)) \end{array}\right\}$$

where $\forall_y$ is read "for all y"

$\Rightarrow$ means "implies"

and $P_i(x,y)$ is the predicate associated with the i^{th} statement of the abstract program.

Q_i; q_{i1}; q_{i2} are the "verification conditions" as defined by Floyd which are associated with the i^{th} statement of the abstract program.

A block diagram of the i^{th} statement of the abstract program is given in Figure E-4, showing the "Floyd verification conditions." If one of the go to statements is replaced by HALT, the corresponding verification condition is replaced by the output predicate $\psi(x,z)$.

The formalism continues by defining two additional well formed formulas

$$T(x) = g^1(x,y) \wedge W^1 \wedge \ldots W_n$$

$\hat{T}(x) = T(x)$ with the output predicate $\psi(x,z)$ replaced by its complement (i.e., $\sim\psi(x,z)$) wherever it appears.

In the above, $\wedge$ means logical "and"
$\sim$ means logical "not"

Finally, the desired result is the two well formed formulas:

$$W_p[\phi,\psi] = V_x\{\phi(x) \Rightarrow T(x)\}$$

$$\hat{W}_p[\phi,\psi] = V_x\{\phi(x) \Rightarrow \hat{T}(x)\}$$

These formulas form the basis for Manna's two theorems (proved in Reference 6):

Theorem 1: The program is partially correct with respect to ϕ and ψ if and only if $W_p[\phi,\psi]$ is satisfiable (i.e., is true under some "interpretation" of the Floyd verification conditions).

Theorem 2: The program is correct with respect to ϕ and ψ if and only if $\hat{W}_p[\phi,\psi]$ is unsatisfiable (i.e., is false under every "interpretation" of the Floyd verification conditions).

Proving satisfiability (or unsatisfiability) of a well formed formula in the predicate calculus is a very complex topic and is not discussed here. The interested reader is referred to References 7 and 8 for an introduction to the topic.

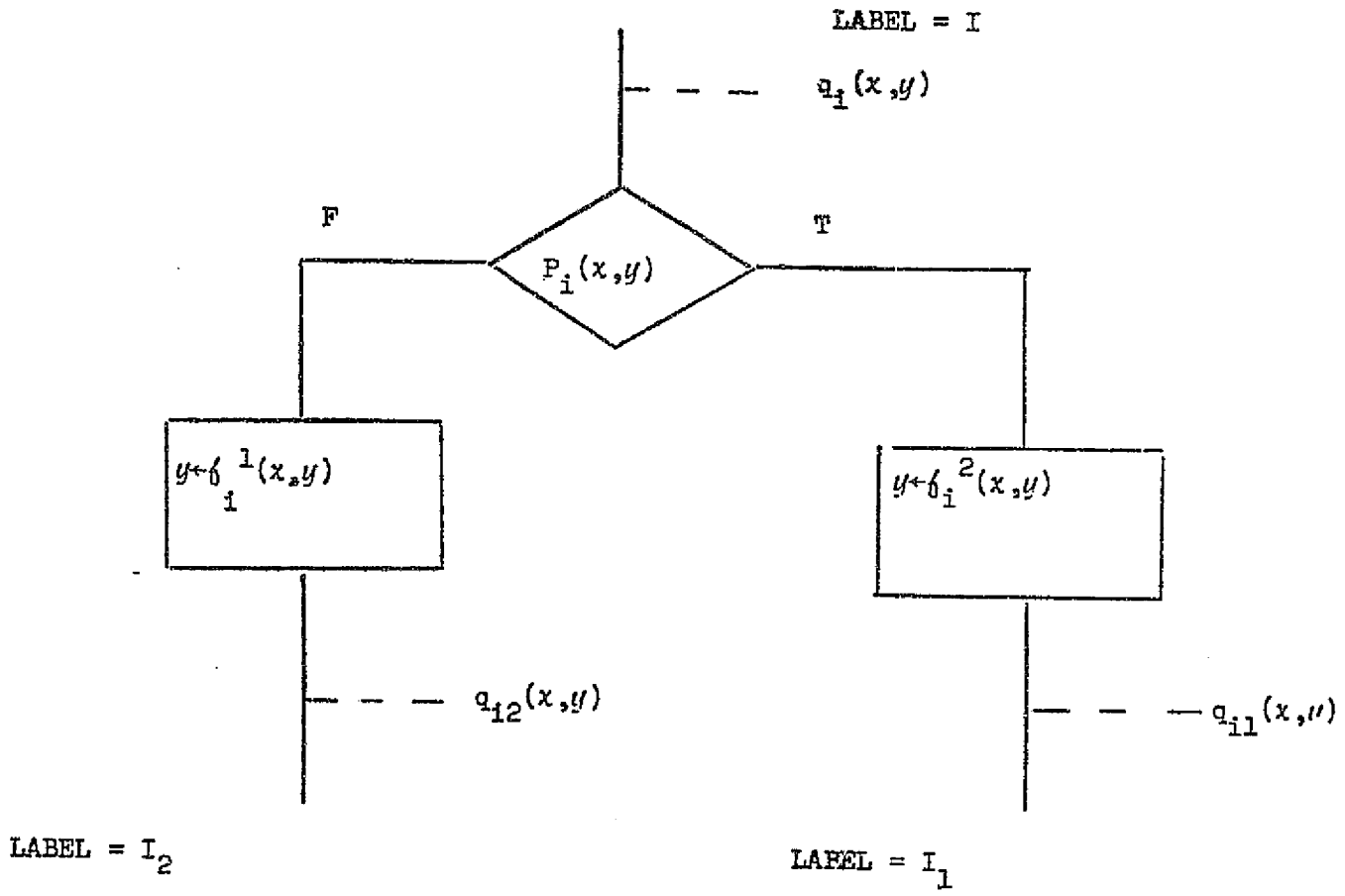


Figure E-4. I^{th} Statement of Abstract Program for Manna's Formalism

REFERENCES

1. P. Naur. Proof of Algorithms by General Snapshots. BIT 6 1966, pp. 310-316.

2. R. W. Floyd. Assigning Meanings to Programs. American Mathematical Society -- Mathematical Aspects of Computer Science, Vol. 19, 1967, pp. 19-32.

3. E. W. Dijkstra. A Constructive Approach to the Problem of Program Correctness. BIT, Vol. 8, No. 3, 1968, pp. 174-186.

4. C.A.R. Hoare. Proof of a Program: FIND. Comm. ACM, January 1971, pp. 39-45.

5. C.A.R. Hoare. An Axiomatic Basis for Computer Programming. COM ACM October 1969, pp. 576-583.

6. Z. Manna. Properties of Programs and the First Order Predicate Calculus. J. ACM, April 1969, pp. 244-255.

7. B. Elspas, K. N. Levitt, R. J. Waldinger, A. Waksman. An Assessment of Techniques for Proving Programs Correct. ACM Computing Surveys, June 1972, pp. 97-147.

8. N. J. Nilsson. Problem Solving Methods in Artificial Intelligence. (Book) McGraw Hill, 1971.

Appendix F

BIBLIOGRAPHY

Adams, D. A., "A Computational Model With Data Flow Sequencing", Stanford University, Computer Science Technical Reports CS-117, December 1968.

Akiyama, F., "An Example of Software System Debugging", Software Engineering Department of Fujitsu Limited, Tokyo.

Allen, C. D., "The Application of Formal Logic to Programs and Programming", IBM Systems Journal, Volume 10, No. 1, 1971.

Allen, C. D., "Derivation of Axiomatic Definitions of Programming Languages from Algorithmic Definitions", Proceedings of the Conference on Proving Assertions About Programs, January 1972.

Amory, W., Clapp, J. A., "A Software Error Classification Methodology", Mitre Corporation Report No. 2648, Volume VII, June 1973.

Ashby, E. T., "The Use of an Auxiliary Computer With a Graphic Display as an On-line Debugging Aid", Naval Postgraduate School Thesis, June 1971.

Ashcroft, E. A., "Program Correctness Methods and Language Definition", Proceedings of the ACM Conference on Proving Assertions About Programs, January 1972.

Ashcroft, E. A., Manna, Z., "The Translation of GOTO Programs to WHILE Programs", Stanford University, Computer Science Report No. CS-188, 1970.

Avizienis, A., "The Methodology of Fault-Tolerant Computing", Software Reliability Course, Engineering 819.59, October 1974.

Avizienis, A., "Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing", International Conference on Reliable Software, April 1975.

Bachman, C. W., "The Programmer as Navigator", Communications of the ACM, Volume 16, No. 11, November 1973.

Baird, G. N., "Program Debugging Using COBOL '74", 1975 National Computer Conference, May 1975.

Baker, F. T., "System Quality Through Structured Programming", AFIPS Conference Proceedings, Volume 41, Part I, 1972.

Baker, F. T., "Chief Programmer Team Management of Production Programming", IBM Systems Journal, Volume II, No. 1, 1972.

Baker, F. T., "Structured Programming in a Production Programming Environment", International Conference on Reliable Software, April 1975.

Balzer, R. M., "EXDAMS - Extendable Debugging and Monitoring System", The Rand Corporation, RM-5772-ARPA, April 1969.

Balzer, R. M., "On the Future of Computer Program Specification and Organization", Rand Corporation, Report R-622-ARPA, AD 731 349, August 1971.

Balzer, R. M., "PORTS-A Method for Dynamic Interprogram Communication and Job Control", Proceedings AFIPS 1971, SJCC, 1972.

Balzer, R. M., "Automatic Programming", Institute Technical Memorandum, USC/Information Sciences Institute, September 1973.

Balzer, R. M., "A Global View of Automatic Programming", Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford Research Institute, 1973.

Balzer, R. M., "A Language-Independent Programmers Interface", Proceedings - AFIPS 1974 NCC.

Bard, Y., "Performance Criteria and Measurement for a Time-Sharing System", IBM Systems Journal, Vol. 10, No. 3, 1971.

Barrett, M. R., "Test Data Generation", U.S. Army Computer Systems Support and Evaluation Command, Washington, D.C., March 1972.

Basili, V. R., and Zelkowitz, M. V., "Compiler Generated Programming Tools", Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Basu, S. K., and Misra, J., "Proving Loop Programs", IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975.

Bauer, F. L., "Software Engineering", Proceedings of IFIP Congress 1971, and in Advanced Course on Software Engineering, New York: Springer-Verlag, 1973.

Benjamin, R. I., Control of the Information System Development Cycle, New York, Wiley, 1971.

Benson, J. P., "Structured Programming Techniques", IEEE Symposium on Computer Software Reliability, May 1973.

Beyer, T., "Preprocessors and Programming Language Reform", Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Beyer, T., "FLECS: User's Manual", University of Oregon Edition, Department of Computer Science, University of Oregon, January 1975.

Birman, A., "On Proving Correctness of Microprograms", IBM Journal, Volume 18, No. 3, May 1974.

Blair, J., "Extendable Non-Interactive Debugging", Debugging Techniques in Large Systems, Prentice-Hall, 1971.

Blevins, P. R., and Ramamoorthy, C. V., "A Classification and Survey of Computer System Performance Evaluation Techniques", University of Texas at Austin, Electronics Research Center Report, April 1970.

Bloom, A. M., "The "ELSE" Must Go, Too", Datamation, May 1975.

Bloom, S., McPheters, M. J., and Tsiang, S. H., "Software Quality Control", IEEE Symposium on Computer Software Reliability, May 1973.

Bochmann, G. V., "Multiple Exits from a Loop Without the GOTO", Communications of the ACM, July 1973.

Boehm, B. W., "Some Information Processing Implications of Air Force Space Missions, 1970-1980", Rand Corporation, Rand Memo RM-6213-PR, January 1970.

Boehm, B. W., "Software and Its Impact: A Quantitative Assessment", Datamation, May 1973.

Boehm, B. W., McClean, R. K. and Urfrig, D. B., "Some Experience With Automated Aids to the Design of Large-Scale Reliable Software", IEEE Transactions on Software Engineering, Volume SE-1, No. 1, March 1975, and International Conference on Reliable Software, April 1975.

Boehm, B. W., "The High Cost of Software", Practical Strategies for Developing Large Software Systems, Addison-Wesley 1975.

Boehm, B. W., "Software Design and Structuring", Practical Strategies for Developing Large Software Systems, Addison-Wesley 1975.

Boettcher, C. B., "Program Evaluator and Tester, CDC User's Manual", McDonnell Douglas Automation Company, Doc. No. M2085074, 1974.

Bohm, C., and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with only Two Formation Rules", Communications of the ACM, Vol. 9, 1966.

Boyer, R. S., Elspas, B., Levitt, K. N., "SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution", International Conference on Reliable Software, April 1975.

Bratman, H. and Court, T., "The Software Factory", Computer May 1975.

Bratman, H., "Automated Techniques for Project Management Control", Practical Strategies for Developing Large Software Systems, Addison-Wesley, 1975.

Bredt, T. H., "A Survey of Models for Parallel Computing", Stanford University, Electronics Lab Report TR-8, August 1970.

Bredt, T. H., and McClusky, E. J., "A Model for Parallel Computer Systems", Stanford University, Electronics Lab Report TR-5, April 1970.

Bredt,T.H.,"Analysis of Operating System Interaction", Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Bright, H. S., and Cole, I. J., "A Method of Testing Programs for Data Sensitivity", Program Test Methods, Prentice Hall, 1973.

Brinch Hansen, P., "The Purpose of Concurrent Pascal", International Conference on Reliable Software, April 1975.

Brooks, F. P., "Testing Computer Programs - Historical Perspective", ACM Sigplan Computer Program Test Methods Symposium, University of North Carolina, June 1972.

Brooks, F. P., "The Mythical Man-Month", International Conference on Reliable Software, April 1975.

Brown, J. R., and Hoffman, R. H., "Automating Software Development, A Survey of Techniques and Automated Tools", TRW, May 1972.

Brown, J. R., "Practical Applications of Automated Software Tools", Proceedings of Western Electronic Show and Convention (WESCON), Los Angeles, California, September 1972.

Brown, J. R., DeSalvio, A. J., Heine, D. E., and Purdy, J. G., "Automated Software Quality Assurance: A Case Study of Three Systems", ACM SIGPLAN Symposium on Computer Program Test Methods, TRW Systems, 1972.

Brown, J. R., Hoffman, R. H., "Evaluating the Effectiveness of Software Verification-Practical Experience with an Automated Tool," TRW Paper #316, FJCC, 1972.

Brown, J. R., DeSalvio, A. J., Heine, D. E., and Purdy, J. G., "Automated Software Quality Assurance", Program Test Methods, Prentice Hall, 1973.

Brown, J. R., "Improving Quality and Reducing Cost of Aeronautical Systems Software Through Use of Automated Tools", TRW Systems Group, Site Defense Program Office.

Brown, J. R., "Why Tools?", Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Brown, J. R., and Lipow, M., "Testing for Software Reliability", International Conference on Reliable Software, April 1975.

Brown, J. R., "Getting Better Software Cheaper and Quicker", Practical Strategies for Developing Large Software Systems, Addison-Wesley, 1975.

Brown, P. J., "Levels of Language for Portable Software", Communications of the ACM, Volume 15, No. 12, December 1972.

Buda, A. O., Granovsky, A. A., and Ershov, A. P., "Implementation of the ALPHA-6 Programming System", International Conference on Reliable Software, April 1975.

Bucher, D.E.W., "Maintenance of the Computer Sciences Teleprocessing System", International Conference on Reliable Software, April 1975.

Buckley, Ltc. F., "Verification of Software Programs", Computers and Automation, February 1971.

Buckley, F. J., "Software Testing - A Report from the Field", IEEE Symposium on Computer Software Reliability, May 1973.

Buechler, J., "A Software Architecture for Sampling Monitors", Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Bullen, R. H., Jr., "Software First Concepts", MITRE Corporation Report No. 2648, Volume III, June 1973.

Burge, W. H., "Combinatory Programming and Combinatorial Analysis", IBM Journal, Volume 16, No. 5, September 1972.

Burkhardt, W. H., "Generating Test Programs from Syntax", Computing, Volume 2, 1967.

Berlakoff, M., "Software Design and Verification System", Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Burstall, R. M., "Proving Properties of Programs by Structural Induction", Computing Journal, Volume 12, No. 1, February 1969.

Burstall, R. M., "Formal Description of Program Structure in First Order Logic", Machine Intelligence 5, American Elsevier, 1970.

Burstall, R. M., "An Algebraic Description of Programs with Assertions, Verification and Simulation", Proceedings of the Conference on Proving Assertions About Programs, January 1972.

Burstall, R. M., "Some Techniques for Proving Correctness of Programs Which Alter Data Structures", Machine Intelligence 7, John Wiley and Sons, 1972.

Burstall, R. M., and Darlington, J., "Some Transformations for Developing Recursive Programs", International Conference on Reliable Software, April 1975.

Buxton, J. N., and Randell, B., "Software Engineering Techniques", Scientific Affairs Division, NATO, Brussels, Belgium, April 1970.

Buxton, J. N., "The Nature and Implications of Software Engineering", The Fourth Generation, Infotech, Ltd., Berkshire, England, 1971.

Buzen, J. P., Chen, P. P., and Goldberg, R. D., "Virtual Machine Techniques for Improving System Reliability", IEEE Symposium on Computer Software Reliability, May 1973.

Cadiou, J. M., and Manna, Z., "Recursive Definitions of Partial Functions and Their Computations", Proceedings of the Conference on Proving Assertions About Programs, January 1972.

Caine, S. H., and Gordon, E. K., "PLL - A Tool for Software Design", 1975 National Computer Conference, May 1975.

Cantrell, H., "Improving Program Reliability Using COTUNE II", Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Caplain, M., "Finding Invariant Assertions for Proving Programs", International Conference on Reliable Software, April 1975.

Carey, L. J., "Software Quality Assurance - A State of the Art Report", Wescon Technical Papers, 1972.

Carlson, G., "How to Save Money With Computer Monitoring", Proceedings 1972 ACM National Conference, New York, 1972.

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

Cerf, V. G., Fernandez, E. B., Gostelow, K. P., and Volansky, S. A., "Formal Control Flow Properties of a Graph Model of Computations", UCLA, Computer Science Report ENG-7178, December 1971.

Cerf, V. G., "Multiprocessors, Semaphorese, and a Graph Model of Computation", UCLA, Computer Science Report ENG-7223, April 1972.

Cerf, V. G., and Estrin, G., "Measurement of Recursive Programs", Proceedings of IFIP Congress 71, Amsterdam: North-Holland 1972.

Chandy, K. M., Brown, J. C., Dissly, C. W., and Uhrig, W. R., "Analytic Models for Rollback and Recovery Strategies in Data Base Systems", IEEE Transactions on Software Engineering, Volume SE-1, No. 1, March 1975.

Chandy, K. M., "A Survey of Analytic Models of Rollback and Recovery Strategies", Computer, May 1975.

Chang, H. Y., Manning, E. G., and Metze, G., Fault Diagnosis of Digital Systems, Wiley-Interscience, 1970.

Cheatham, T. E., Jr., "On A Laboratory of the Study of Automatic Programming", ACM Conference on Proving Assertions About Programs, 1972.

Cheng, L. L., and Sullivan, J. E., "Case Studies in Software Design", Mitre Corporation Report, MTR-2874, Volume I, June 1974.

Cheng, L. L., "Some Case Studies in Structured Programming", MITRE Corporation Report No. MTR-2648, Vol. VI, June 1973.

Chirica, L. M., and Martin, D. F., "An Approach to Compiler Correctness", International Conference on Software Reliability, April 1975.

Cicu, A., Maiocchi, M., Polillo, R., Sardoni, A., "Organizing Tests During Software Evolution", International Conference on Reliable Software, April 1975.

Clapp, J. A., LaPadula, L. J., "Engineering of Quality Software Systems", Mitre Corporation Report MTR-2648, Volume I, June 1973.

Clapp, J. A., Sullivan, J. E., SIMON: Finding the Answers to Software Development Problems, Mitre Corporation Report No. MTR-152, May 1974.

Clark, L., "A System to Generate Test Data and Symbolically Execute Programs", Report #CU-CS-060-75, February 1975, Department of Computer Sciences, University of Colorado, Boulder, Colorado.

Clint, M., "Program Proving", Coroutines Acta Information 2, 1973.

Clint, M., and Hoare, C.A.R., "Program Proving: Jumps and Functions", Acta Informatica, Volume 1, No. 3, 1972.

Cody, J. W., "The Evaluation of Mathematical Software", Program Test Methods, Prentice Hall, 1973.

Cohen, J., and Zuckerman, C., "Two Languages for Estimating Program Efficiency", Communications of the ACM Volume 17, No. 6, June 1974.

Conrow, K., and Smith, R. G., "NEATER 2--A PL/I Source Statement Reformatter", Communications of the ACM, November 1970.

Constable, R. L., "Constructive Mathematics and Automatic Program Writers", Proceedings of IFIP Congress 71, Amsterdam: North-Holland 1972.

Conway, R., and Gries, D., An Introduction to Programming : A Structured Approach Using PL/1 and PL/C, Cambridge, Mass., Winthrop Publishers 1973.

Cooper, D. C., "Programs for Mechanical Program Verification" University College of Swansea, Computer Science Memorandum No. 13, July 1970.

Corrigan, A. E., "Results of an Experiment in the Application of Software Quality Principles", Mitre Corporation Report, MTR-2874, Vol. III, June 1974.

Coutinho, J. de S., "Software Reliability Growth", IEEE Symposium on Computer Software Reliability, May 1973.

Crocker, S., and Balzer, R., "The National Software Works: A New Distribution System for Software Development Tools", Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Culpepper, L. M., "A System for Reliable Engineering Software", International Conference on Reliable Software, April 1975.

Daly, D., "Overview of Performance Measurement Techniques," SIGCOSIM Newsletter, No. 8, Part II, April 1971.

Davis, R.M., "Standards for Software - What is in the Future," presented at ADAPSO Software Section Management Conference, Dallas, Texas, February 1972.

Davis, R.M., "Quality Software Can Change the Computer Industry," Program Test Methods, Prentice Hall, 1973.

de Balbine, G., "Using the FORTRAN Structuring Engine," Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

de Balbine, G., "Tools for Modern FORTRAN Programming," Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

de Balbine, G., "Better Manpower Utilization Using Automatic Restructuring," 1975 National Computer Conference, May 1975.

Dennis, J.B., "The Design and Construction of Software Systems," Advanced Course on Software Engineering, New York: Springer-Verlag, 1973.

Dennis, J.B., "Concurrency in Software Systems," Advanced Course on Software Engineering, New York: Springer-Verlag, 1973.

Dennis, J.B., "Modularity," Advanced Course on Software Engineering, New York: Springer-Verlag, 1973.

DeRemer, F., Kron, H., "Programming-in-the-Large Versus Programming-in-the-Small," International Conference on Reliable Software, April 1975.

DeRoze, B.C., "Software Reliability via the Specification - A Quantitative Approach," submitted to the International Conference for Reliable Software, 1975.

DeRoze, B.C., "Survey of Software Verification/Validation Technology," paper presented at ACM Conference, San Diego, 1974 (viewgraphs also).

Deutsch, L.P., "An Interactive Program Verifier," Ph.D., dissertation, Department of Computer Science, University of California, Berkeley, Calif., May 1973.

DeViot, A.R., "The PRO/TEST Library of Testing Software," Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Dickson, J., Hesse, J., Kientz, A., Shooman, M., "Quantitative Analysis of Software Reliability," 1972 Annual Reliability Symposium, IEEE, January 1972.

Dijkstra, E.W., "Programming Considered as a Human Activity," Proceedings of the IFIP Congress, 1965.

Dijkstra, E.W., "GOTO Statement Considered Harmful," Communications of the ACM, Volume II, No. 3, March 1968.

Dijkstra, E.W., "A Constructive Approach to the Problem of Program Correctness," BIT, Volume 8, No. 3, 1968.

Dijkstra, E.W., "The Structure of the "THE" - Multiprogramming System," Communications of the ACM, Vol. 11, No. 5, May 1968.

Dijkstra, E.W., "Structured Programming," Software Engineering Techniques, NATO Science Committee, 1969.

Dijkstra, E.W., "Notes on Structured Programming," Technische Hogeschool Eindhaven (THE), 1969.

Dijkstra, E.W., "Concern for Correctness as a Guiding Principle for Program Composition," The Fourth Generation, Infotech, Ltd., Berkshire, England, 1971.

Dijkstra, E.W., "Guarded Commands, Non-determinancy and a Calculus for the Derivation of Programs",International Conference on Reliable Software, April 1975.

Dijkstra, E.W., "Correctness Concerns and, Among Other Things, Why They are Resented," International Conference on Reliable Software, April 1975.

Early, J., "Toward an Understanding of Data Structures," Communications of the ACM, Vol. 14, No. 10, October 1971.

Edwards, N.P., "The Effect of Certain Modular Design Principles on Testability," International Conference on Reliable Software, April 1975.

Ehrman, J.R., "System Design, Machine Architecture, and Debugging," SIGPLAN Notices, Volume 7, No. 8, August 1972.

Ellingson, O.E., "Computer Program and Change Control," IEEE Symposium on Computer Software Reliability, May 1973.

Elmendorf, W.R., "Controlling the Functional Testing of an Operating System," IEEE Transactions System Science and Cybernetics, SSC-5, October 1969.

Elmendorf, W.R., "Disciplined Software Testing," Debugging Techniques in Large Systems, Prentice-Hall, 1971.

Elspas, B., Green, M.W., and Levitt, K.N., "Software Reliability Computer," (Computer Group News), January-February 1971.

Elspas, B., Green, M.W., Levitt, K.N., and Waldinger, R.J., "Research in Interactive Program Proving Techniques," SRI Report 8398-II, Stanford Research Institute, Menlo Park, Ca., 1972.

Elspas, B., Levitt, K.N., Waldinger, R.J., and Waksman, A., "An Assessment of Techniques for Proving Program Correctness," Computing Surveys, Vol. 4, No. 2, June 1972.

Elspas, B., Levitt, M.W., and Waldinger, R.J., "An Interactive System for the Verification of Computer Programs," Final Report, SRI Project 1891, Stanford Research Institute, Menlo Park, Ca., 1973.

Elspas, B.,"The Semi-Automatic Generation of Inductive Assertions for Proving Program Correctness," Interim Report, SRI Project 2686, Stanford Research Institute, Menlo Park, Ca., 1974.

Endres, A., "An Analysis of Errors and Their Causes in System Programs," International Conference on Reliable Software, April 1975.

Engelman, C., "Towards an Analysis of the LISP Programming Language," Mitre Corporation Report No. 2648, Vol. IV, June 1973.

Estep, J.G., "A Software Availability and Reliability Model," IEEE Symposium on Computer Software Reliability, May 1973.

Evans, R.V., "Multiple Exits from a Loop Using Neither GOTO nor Labels," Communications of the ACM, November 1974.

Fairley, R.E., "An Experimental Program Testing Facility," Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Feldman, J.A., "Toward Automatic Programming," Software Engineering Techniques, Scientific Affairs Division, NATO, Brussels, 1970.

Fleischer, R.J., "Effects of Management Philosophy on Software Production," Mitre Corporation Report MTR-2648, Vol. II, June 1973.

Florentin, J.J., "Flow Analysis for Program Correctness," University of Waterloo, CSRR 2054 Research Report, 1970.

Floyd, R.W., "Nondeterministic Algorithms," Journal of the ACM, Vol. 14, No. 4, October 1967.

Floyd, R.W., "Assigning Meanings to Programs," Mathematical Aspects of Computer Science, Vol. XIX, American Mathematical Society, Providence, R.I., 1967.

Floyd, R.W., "Toward Interactive Design of Correct Programs," Proceedings of IFIP Congress 71, Amsterdam: North-Holland, 1972.

Flynn, R.J., "On the Smallest Number of Program Modules Needed to Duplicate Dynamic Independent Interactions," IEEE Symposium on Computer Software Reliability, May 1973.

Forsythe, A.B., "Adequacy and Validity of Statistical Analysis," Computer Science and Statistics: 8th Annual Symposium on the Interface, Febraury 1975.

Fosdick, L.D., "BRANL, A FORTRAN Program to Identify Basic Blocks in FORTRAN Programs," Report #CU-CS-040-74, Dept. of Computer Science, University of Colorado, March 1974.

Fosdick, L.D., and Osterweil, L.J., "DAVE - A FORTRAN Program Analysis System," Computer Science and Statistics: 8th Annual Symposium on the Interface, February, 1975.

Fragola, J.R., and Spahn, J.F., "The Software Error Effects Analysis: A Qualitative Design Tool," IEEE Symposium on Computer Software Reliability, May 1973.

Freeman, P., "A Model for Functional Reasoning in Design," Proceedings, Second International Joint Conference on Artificial Intelligence, London, 1971.

Freeman, P., "Functional Programming Testing and Machine Aids," Program Test Methods, Prentice Hall, 1973.

Freeman, P., "Software Engineering Bibliography," Rough Draft, ICS Department, University of California, Irvine, Ca., September, 1974.

Freeman, P., "Toward Improved Review of Software Designs," 1975 National Computer Conference, May 1975.

Fujii, R.U., and Hartwick, R.D., "Test Techniques for Large-Scale Program," Logicon, January 1974.

C.3

Gaines, R. S., "Compiler Construction for Debugging", Debugging Techniques in Large Systems, Prentice Hall, 1971.

Gannon, J. D., and Horning, J. J., "The Impact of Language Design on the Production of Reliable Software", International Conference on Reliable Software, April 1975.

Garland, S. J., and Luckham, D. C., "Translating Recursion Schemes Into Program Schemes", Proceedings of the Conference on Proving Assertions About Programs, January 1972.

General Research Corporation, "RXVP-1 User's Guide", February 1975.

Gentleman, W. M., and Wichmann, B. A., "Timing on Computers", SIGARCH 2, October 1973.

Gerhart, S. L., "Knowledge About Programs: A Model and a Case Study", International Conference on Reliable Software, April 1975.

German, S. M., and Wegbreit, B., "A Synthesizer of Inductive Assertions", IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975.

Gesche, C. M., and Mitchell, J. G., "On the Problem of Uniform References to Data Structures", International Conference on Reliable Software, April 1975.

Gibson, C. G., and Railing, L. R., "Verification Guidelines", TRW SS-71-04, TRW Software Series, August 1971.

Girard, E., and Rault, J. C., "A Programming Technique for Software Reliability", IEEE Symposium on Computer Software Reliability, May 1973.

Glassman, B. A., and Bonham, G. P., "Automating Software Development", Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Goldberg, J., "Toward Better Software", Electronics, Vol. 44, No. 19, September 1971.

Gomory, R. E., "An Algorithm for Integer Solutions to Linear Programs", Recent Advances in Mathematical Programming, McGraw-Hill, N.Y., 1963.

Good, D. I., "Toward a Man-Machine System for Proving Program Correctness", Ph.D., dissertation, Dept. of Computer Science, University of Wisconsin, Madison, Wisconsin, June 1970.

Good, D. I., "Developing Correct Software," Proceeding of the First Texas Conference on Computer Systems, June 1972.

Good, D. I., and Raglund, L. C., "Nucleus-A Language of Provable Programs", Program Test Methods, Prentice Hall, 1973.

Good, D. I., "Provable Programs and Processors", National Computer Conference, 1974.

Good, D. I., "Provable Programming", International Conference on Reliable Software, April 1975.

Good, D. I., London, R. L. and Bledsoe, W. W., "An Interactive Program Verification System", IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975, and International Conference on Reliable Software, April 1975.

Goodenough, J. B., and Eanes, R. S., "Program Testing and Diagnosis Technology", SOFTECH Report to Frankford Arsenal, April 1973.

Goodenough, J. B., and Gerhart, S. L., "Toward a Theory of Test Data Selection", International Conference on Reliable Software, April 1975.

Goodman, L. I., "Complexity Measures for Programming Languages", Mass. Institute of Technology, AD 729-001, September 1971.

Goodnight, J. H., "Validity Checking - How Far Should We Go?" Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Goos, G., "Hierarchies", Advanced Course on Software Engineering New York: Springer-Verlag, 1973.

Goos, G., "Language Characteristics", Advanced Course on Software Engineering, New York: Springer-Verlag, 1973.

Goos, G., "Documentation", Advanced Course on Software Engineering, New York: Springer-Verlag, 1973.

Gostelow, K. P., "Flow of Control, Resource Allocation and the Proper Termination of Programs", UCLA Computer Science Report ENG-7179, December 1971.

Gotlieb, C. C., and MacEwen, G. H., "System Evaluation Tools", NATO Working Conference on Software Engineering, Brussels, Belgium, NATO 1970.

Gotlieb, C. C. "Performance Measurement", Advanced Course on Software Engineering, New York: Springer-Verlag, 1973.

Graham, P. M., "Performance Prediction", Advanced Course on Software Engineering, New York: Springer-Verlag, 1973.

Graham, R. M., Clancy, G. J., Jr., and DeVaney, D. B., "A Software Design and Evaluation System", Communications of the ACM, February 1973.

Green, E., "What, How and When to Test", Workshop on Currently Available Test Tools - Technology and Experience, April 1975.

Gries, D., "Programming by Induction", Information Processing Letters, Vol. 1, No. 3, February 1972.

Grishman, R., "The Debugging System - AIDS", Proceedings of the SJCC, 1970.

Grishman, R., "Criteria for a Debugging Language", Debugging Techniques in Large Systems, Prentice-Hall, 1971.

Gruenberger, F., "Program Testing and Validating", Computing: A First Course, 1968.

Gruenberger, F., "Program Testing: The Historical Perspective", Program Test Methods, Prentice Hall, 1973.

Hall, A.D., and Ryder, B.G., "The PFORT Verifier-Installation and Maintenance," Bell Laboratories, Murray Hill, New Jersey.

Hall, A.D., and Ryder, B.G., "The PFORT Verifier," Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Haney, F.M., "Module Connection Analysis - A Tool for Scheduling Software Debugging Activities," Proceedings of the FJCC, 1972.

Hanford, K.V., "Automatic Generation of Test Cases," IBM Systems Journal, No. 4, 1970.

Hansen, P.B., "Testing Multiprogramming Systems," Software Practice and Experience, April-June 1973.

Harper, W.L., Data Processing Documentation: Standards, Procedures and Applications, Prentice-Hall, 1973.

Hartwick, R.D., "Verification and Validation," Logicon, January 1974.

Hartwick, R.D., Fujii, R.U.,"Addendum to Software Reliability Sample Verification and Validation Effort," Logicon, October 1974.

Hecht, H., "The Scope of Software Reliability," Overview, Software Reliability Course, Engineering 819.59, UCLA, October 1974.

Hecht, H., "Economics of Reliability and Related Subjects," Software Reliability Course, Engineering 819.59, UCLA, October 1974.

Helms, H.J., "Evaluation in the Computing Center Environment," Advanced Course Software Engineering, New York: Springer-Verlag, 1973.

Henderson, P., and Snowdon, R., "An Experiment in Structured Programming," BIT 12, 1972.

Henderson, P., "Finite State Modeling in Program Development," International Conference on Reliable Software, April 1975.

Henderson, V.D., "Program Validation," Logicon, Inc., San Pedro, California, (GUESS) 1970.

Hennell, M.A., "Experimental Test Bed for Numerical Software," Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Hetzel, W.C., "Principles of Computer Program Testing," Program Test Methods, Prentice Hall, 1973.

Hetzel, W. C., "A Definitional Framework," and other Illustrations for the Software Reliability Course, Engineering 819.59, UCLA, October 1974.

Hewitt, C.E., and Smith, B., "Towards a Programming Apprentice," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975.

Hill, I.D., "Faults in Functions in ALGOL and FORTRAN," Computer Journal, Vol. 14, August 1971.

Hoare, C.A.R., "An Axiomatic Approach to Computer Programming," Communication of the ACM, Vol. 12, No. 10, October 1969.

Hoare, C.A.R., "Proof of a Program: FIND," Communications of ACM, Vol. 14, No. 1, 1971.

Hoare, C.A.R., "Proof of Correctness of Data Representations," Acta Informatica 1, Springer-Verlag, 1972.

Hoare, C.A.R., "Proof of a Structured Program: The Sieve of Eratosthenes," the Computer Journal, Vol. 15, No. 4, 1972.

Hoare, C.A.R., "The Quality of Software," Software--Practice and Experience, Vol. 2, 1972.

Hoare, C.A.R., "A Note on the FOR Statement," BIT, Vol. 12, 1972.

Hoare, C.A.R., "Data Reliability," International Conference on Reliable Software, April 1975.

Hoffman, R.H., "Automated Verification System User's Guide," TRW Note #72-FMT-8 Project Apollo, Task MSC/TRW A-527, January 1972.

Hoffman, R.H., "Automated Verification System: Test Data Effectiveness Measurement Subsystem User's Guide," TRW Systems Group, for NASA Johnson Space Center, Houston, Texas 1974.

Hoffman, R.H., "NASA/Johnson Space Center Approach to Automated Test Data Generation," Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Holland, J.G., "Acceptance Testing for Applications Programs," Program Test Methods, Prentice Hall, 1973.

Holton, J.B., and Bryan, B., "Structured Top-Down Flowcharting, Datamation, May 1975.

Hopcroft, J., and Tarjan, R., "Efficient Algorithms for Graph Manipulation," Stanford Unviersity Report No. STAN-CS-71-207, AD 72 6169, March 1971.

Hopwood, M.D., and Lockett, J., "Experience with the RAND Monitor/Stimulator," Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Horning, J.J., and Randell, B., "Structuring Complex Processes," IBM T.J. Watson Research Center, Report RC 2459, May 1969.

Horowitz, E., "FORTRAN, Can It Be Structured and Should It Be?", Practical Strategies for Developing Large Software Systems, Addison-Wesley, 1975.

Howard, J.H., and Alexander, W.P., "Analyzing Sequences of Operations Performed by Programs," Program Test Methods, Prentice Hall, 1973.

Howden, W.E., "Methodology for the Automatic Generation of Program Test Data," McDonnell Douglas Technical Report #41, February 1974.

Howden, W. C., "Proving Correctness by Testing," November 1974.

Howden, W.E., "Methodology for the Generation of Program Test Data," Research Paper, McDonnell Douglas, May 1975.

Howden, W.E., Stucki, L.G., "Methodology for the Effective Test Case Selection," Final Report, MDAC-W, MDC G5301.

Howden, W.E., Laub, J., "Automatic Case Analysis of Programs," Computer Science and Statistics, 8th Annual Symposium on the Interface, February 1975.

Howden, W.E., "Systems for Automating the Generation of Program Test Data," Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Hughes, K., Binns, J., Cooke, A., "Keeping in Tune," Data Processing, July-August 1974.

Hull, T.E., Enright, W.H., and Sedgwich, A.E., "The Correctness of Numerical Algorithms," Proceedings of the Conference on Proving Assertions About Programs, January 1972.

Information Research Associates, "Reliability Techniques for Computer Executive Programs," Summary Report NAS8-2666-9.

Ingalls, D.H., "FETE--A FORTRAN Execution Time Estimator," Stanford University, Computer Science Report 204, 1971.

Ingalls, D., "The Execution Time Profile as a Programming Tool," Compiler Optimization, R. Rustin (ed.), 2nd Courant Computer Science Symposium, 1970, Prentice-Hall, 1972.

Itoh, D., and Izutani, T., "FADEBUG-I, A New Tool for Program Debugging," IEEE Symposium on Computer Software Reliability, May 1973.

IBM, "HIPO: Design Aid and Documentation Tool," IBM Audio Education Course Form No. SR20-9413.

IBM, "Chief Programmer Teams Principles and Procedures," IBM, Gaithersburg, Maryland, June 1971.

IBM, "Test IMS Utilities," Program Description/Operations Manual, SH 20-1307-0.

IBM, "How to Write Correct Programs and Know It," IBM, Gaithersburg, Maryland, February 1973.

Ikezawa, M.A., "AMPIC," Workshop on Currently Available Program Testing Tools: Technology and Experience, April 1975.

Infante, R., and Montanari, U., "Proving Structured Programs Correct, Level by Level," International Conference on Reliable Software, April 1975.

Jackson, M., and Swanwick, A.B., "Segmented-Level Programming," Computers and Automation, February 1969.

Jackson, R.S., and Bravdica, S.A., "Software Validation of the Titan III C Digital Flight Control System Utilizing a Hybrid Computer," Proceedings of the FJCC, 1971.

Jacobs, W., "A Structure for Systems that Plan Abstractly," Proceedings of the AFIPS, 1971 SJCC, 1971.

James, E.B., and Partridge, D.P., "Adaptive Correction of Program Statements," Communications of the ACM, Vol. 16, No. 1, January 1973.

Jelinski, Z., and Chung, G.S., "Generalized Events-Oriented Simulation System (GESS), A Performance Evaluation Tool," Proceedings of the Computer Performance Evaluation Users Group, October 1972, Washington, D.C.

Jelinski, Z., and Moranda, P., "Software Reliability Research," Statistical Computer Performance Evaluation, Academic Press, 1972.

Jelinski, Z., and Moranda, P.B., "Applications of a Probability Based Model to A Code Reading Experiment," IEEE Symposium on Computer Software Reliability, May 1973.

Jelinski, Z., "Can Statistics Be Applied to Software Reliability-Historical Perspective," Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Jones, C.B., "Formal Development of Correct Algorithms: An Example Based on Earley's Recognizer," Proceedings of the ACM Conference on Proving Assertions About Programs, January 1972.

Kane, J.R., and Yau, S.S., "Concurrent Software Fault Detection," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975.

Kaplan, D.M., "Proving Things About Programs," Fourth Annual Princeton Conference on Information Sciences and Systems, May 1970.

Karnes, R.E., and Carter, W.A., "Computer Design Verification Via Software Simulation," National Computer Conference, May 1975.

Katz, S.M., and Manna, Z., "A Heuristic Approach to Program Verification," Proceedings IFCAI-73, August 1973.

Katz, S., Manna, Z., "Towards Automatic Debugging of Programs," International Conference on Reliable Software, April 1975.

Keezer, E.I., "Practical Experiences in Establishing Software Quality Assurance," IEEE Symposium on Computer Software Reliability, May 1973.

Keirstead, R.E., and Parker, D.B., "On the Feasibility of Formal Certification," Program Test Methods, Prentice Hall, 1973.

Kernighan, B.W., and Plaugher, P.J., "Programming Style for Programmers and Language Designers," IEEE Symposium on Computer Software Reliability, May 1973.

Kimbleton, S.R., and Moore, C.G., "A Probabilistic Framework for System Performance Evaluation," Proceedings of the ACM SIGOPS Workshop on System Performance Evaluation, 1971.

Kimbleton, S.R., "The Role of Computer System Models in Performance Evaluation," Communications of the ACM, Vol. 15, No. 7, July 1972.

Kimbleton, S.R., "A Heuristic Approach to Computer Systems Performance Improvements, I-A Fast Performance Prediction Tool," National Computer Conference, May 1975.

King, J.C., "A Program Verifier, Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, Pennsylvania, September 1969.

King, J.C., "Proving Programs to be Correct," IEEE Transactions and Computers, November 1971.

King, J.C., "A Verifying Compiler," Debugging Techniques in Large Systems, Prentice Hall, 1971.

King, J.C., and Floyd, R.W., "An Interpretation Oriented Theorem Prover Over Integers," Journal of Computer and System Sciences, Vol. 6, No. 4, August 1972.

King, J.C., "Abstract Machines and Software Design," SIGPLAN Notices, Vol. 8, No. 9, September 1973.

King, J.C., "A New Approach to Program Testing," 1975 International Conference on Reliable Software, April 1975.

King, N.J., "Testing Conversational Systems," Debugging Techniques in Large Systems, Prentice Hall, 1971.

Kirchoff, M.K., and Ryan, R.H., "The Need to Salvage Test Tool Technology," Workshop on Currently Available Program Testing Tools: Technology and Experience, April 1975.

Kirchoff, M.K., and Fee, J.B., Software Development Standards and Conventions Document, McDonnell Douglas Astronautics Company, June 1974.

Kling, R.E., "Towards a Person-Centered Computer Technology," Proceedings of the ACM National Conference, New York, 1973.

Knuth, D.E., Floyd, R.W., "Notes on Avoiding GOTO Statements," Stanford University, Computer Science Technical Report CS-148, January 1970.

Knuth, D.E., "An Empirical Study of FORTRAN Programs," Stanford University, Computer Science Technical Report CS-186, 1971.

Knuth, D.E., "A Review of Structured Programming," Stanford University, Computer Science Technical Report CS-371, June 1973.

Knuth, D.E., and Stevenson, F.R., "Optimal Measurement Points for Program Frequency Counts," BIT 13, 1973.

Koffman, E.B., and Blount, S.E., "Artificial Intelligence and Automatic Programming in CAI," Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford Research Institute, 1973.

Kolence, K.W., "A Software View of Measurement Tools," Datamation, January 1971.

Kolence, K.W., "Software Techniques," SIGCOSIM Newsletter, No. 8, Part II, April 1971.

Kolence, K.W., "Software Physics and Computer Performance Measurements," Proceedings 1972 ACM National Conference, New York, 1972.

Kolence, K.W., "Experiments and Measurements in Computing," 1st Annual SIGME Symposium on Measurement and Evaluation, New York, 1973.

Kolence, K.W., "Software Physics," Datamation, June 1975.

Kopetz, H., "On the Connections Between Range of Variable and Control Structure Testing," International Conference on Reliable Software, April 1975.

Kosajaru, S.R., "Correctness of Programs - Writing Correct Programs," Concepts in Quality Software Design, NBS Technical Note 842, 1972.

Kosajaru, S.R., "Structured Programs," Concepts in Quality Software Design, NBS Technical Note 842, 1972.

Kosajaru, S.R., and Ledgard, H.F., "Perspectives on Quality Software," Concepts in Quality Software Design, NBS Technical Note 842, 1972.

Kosajaru, S.R., "Analysis of Structured Programs," Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, New York, 1973.

Kosy, Donald K., "Approaches to Improved Program Validation Through Programming Language Design," Program Test Methods, Prentice Hall, 1973.

Krause, K.W., et.al., "Optimal Software Test Planning Through Automated Network Analysis," IEEE Symposium on Computer Software Reliability, May 1973.

Kuhn, H.W., "Solvability and Consistency for Linear Equations and Inequalities," American Mathematical Monthly, April 1956.

Kulsrud, H.E., "Extending the Interactive Debugging System-HELPER," Courant Computer Science Symposium 1 June 1970; Debugging Techniques in Large Systems, Prentice-Hall, 1971.

La Padula, L.J., "Software Reliability Modeling and Measurement Techniques," MTR 2648, Vol. VIII, The Mitre Corporation, Bedford, Mass., 1973.

Larmouth, J., "Serious FORTRAN," Software Practice and Experience, Vol. 3, No. 2, April-June 1973.

Larsen, G.H., "Software: A Qualitative Assessment or the Man in the Middle Speaks Back," Datamation, November 1973.

Laventhal, M.S., "Verifying Programs Which Operate on Data Structures," International Conference on Reliable Software, April 1975.

Leavenworth, B.M., ed., "Control Structures in Programming Languages," SIGPLAN Notices, Vol. 7, No. 11, November 1972.

Leavenworth, K., "Modular Design of Computer Programs," Data Management, July 1974.

Ledgard, H.F., "The Case for Top-Down Programming," Concepts in Quality Software Design, NBS Technical Note 842, 1972.

Ledgard, H.F., "Towards a Formalization for Quality Software," Concepts in Quality Software Design, NBS Technical Note 842, 1972.

Ledgard, H.F., "The Case for Structured Programming," BIT, Vol. 14, 1974.

Lee, J.A.N., "The Definition and Validation of the Radix Sorting Technique," Proceedings of the Conference on Proving Assertions About Programs, January 1972.

Lemoine, M., and Y. Rousselot, J., "A Tool for Debugging FORTRAN Programs," Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Lester, B.P., "Cost Analysis of Debugging Systems," MIT Report MAC-TR-90, AD 730 521, September 1971.

Linden, T.A., "A Summary of Progress Toward Proving Program Correctness," Proceedings AFIPS 1972 FJCC, 1972.

Lipow, M., "Maximum Likelihood Estimation of Parameters of Software Time-to-Failure Distribution," TRW Systems Group 2260.1.9-73B-15, Revision 1, 1973.

Lipow, M., "Some Directed Graph Methods for Analyzing Computer Programs," Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Liskov, B.H., and Towster, E., "The Proof of Correctness Approach to Reliable Systems," Mitre Corporation Report MTR-2073, July 1971.

Liskov, B.H., "A Design Methodology for Reliable Software Systems," Proceedings of Fall Joint Computer Conference, AFIPS, Vol. 41, Part I, 1972.

Liskov, B.H., "Guidelines for the Design and Implementation of Reliable Software Systems," Mitre Corp., Report No. MTR-2345, April 1972.

Liskov, B.H., and Zilles, S.N., "Specification Techniques for Data Abstractions," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975 and International Conference on Reliable Software, April 1975.

Liskov, B.H., "Data Types and Program Correctness," (position paper), National Computer Conference, May 1975.

Lite, S., "Using a System Generator," Datamation, June 1975.

Littlewood, B., and Verrall, J.L., "A Bayesian Reliability Growth Model for Computer Software," IEEE Symposium on Computer Software Reliability, May 1973.

Littlewood, B., "A Reliability Model for Markov Structured Software," International Conference on Reliable Software, April 1975.

Littrell, R.F., "A Step Toward Quality Control in Computer Programming: Understanding the Psychology of the Management of Computer Programmers," Proceedings of the 1973 National Conference, New York, 1973.

Llewelyn and Wilkens, "The Testing of Computer Software," Software Engineering NATO Science Affairs Division, Brussels 39, January 1969.

London, R.L., "Bibliography on Proving the Correctness of Computer Programs," University of Wisconsin Computer Sciences Dept., Technical Report #64, Madison, Wisconsin, October 1969.

London, R.L., "Computer Programs Can be Proved Correct," Theoretical Approaches to Problem Solving, Vol. 28, Lecture Notes in Operations Research and Mathematical Systems, Springer-Verlag, 1970.

London, R.L., "Proof of Algorithms - A New Kind of Certification," Communications of the ACM, June 1970.

London, R.L., "Proving Programs Correct: Some Techniques and Examples," BIT, Vol. 10, No. 2, 1970.

London, R.L., "Certification of Algorithm 245 Treesort 3: Proof of Algorithms - A New Kind of Certification," Communications of the ACM, Vol. 13, 1970.

London, R.L., "Software Reliability Through Proving Programs Correct," International Symposium on Fault Tolerant Computing, March 1971.

London, R.L., "Correctness of a Compiler for a LISP Subset," Proceedings of the ACM Conference on Proving Assertions About Programs, January 1972.

London, R.L., "Program Verification (correctness proofs)," Software Reliability Course, Engineering 819.59, UCLA, October 1974.

London, R.L., "A View of Program Verification," International Conference on Reliable Software, April 1975.

Lowe, T.C., "Automatic Segmentation of Cyclic Program Structures Based on Connectivity and Processor Timing," Communications of the ACM, January 1970.

Lucas, H.C., Jr., "Synthetic Program Specifications for Performance Evaluation," Proceedings of 1972 ACM National Conference, 1972.

Lucena, C.J., (abstract only), "A Methodology for Producing Reliable Software Systems," Proceedings of the 1973 ACM National Conference, 1973.

Luckham, D.C., Park, D.M.R., and Paterson, M.S., "On Formalized Computer Programs," Journal of Computer and Systems Sciences, June 1970.

Lynch, W.C., Langer, J.W., Schwartz, M.S., "Reliability Experience with CHI/OS," International Conference on Reliable Software, April 1975.

Lyon, G.E., "Static Language Analysis," National Bureau of Standards, Technical Note 797, 1973.

Lyons, T., and Bruno, J., "An Interactive System for Program Verification," Proceedings of the Symposium on Computers, Polytechnic Institute of Brooklyn, April 1971; also Princeton University, Electrical Engineering Department, Report No. 91, May 1971.

MacWilliams, W. H., "Reliability of Large Real-Time Control Software", IEEE Symposium on Computer Software Reliability, May 1973.

Madden, R. L., "Software Accounting and the Hardware Monitor: Their Marriage in Performance Analysis", Proceedings of the 1972 ACM National Conference, 1972.

Manna, Z., "The Correctness of Programs", Journal of Computer and System Sciences, Vol. 3, No. 2, May 1969.

Manna, Z., "Mathematical Theory of Partial Correctness", Stanford University, Computer Science Technical Report, 1970.

Manna, Z., and McCarthy, J., "Properties of Programs and Partial Function Logic", Machine Intelligence 5, American Elsevier Publishing Company, 1970.

Manna, Z., and Pnueli, A., "Formalization of Properties of Functional Programs", Journal of ACM, Vol. 17, No. 3, July 1970.

Manna, Z., and Waldinger, R. J., "Toward Automatic Program Synthesis", Communications of the ACM, Vol. 14, No. 3, March 1971.

Manna, Z., Ness, S., and Vuillen, J., "Inductive Methods for Proving Properties of Programs", Proceedings of the Conference on Proving Assertions About Programs, January 1972.

Marshall, J. J., "New Approaches to Documentation and Debugging", Data Processing, Vol. 14, 1970.

Martin, J. J., "Generalized Structured Programming", Proceedings of the 1974 National Computer Conference, 1974.

McClusky, E. J., "Test and Diagnosis Procedures for Digital Networks", Computer, Vol. 4, No. 1, January - February 1971.

McCracken, D. D., "International Conference on Reliable Software", Datamation, June 1975.

McGeachie, J. S., "Reliability of the Dartmouth Time Sharing System", IEEE Symposium on Computer Software Reliability, May 1973.

McGowen, C., "The Most Recent Error - Its Causes and Correction", Proceedings on the Conference on Proving Assertions About Programs, January 1972.

McGowen, C. L., Kelly, J. R., Top-Down Structured Programming Techniques, Petrocelli/Charter New York 1975.

McHenry, R. C., "Management Concepts for Top-Down Structured Programming", IBM Corporation, February 1973.

McKeeman, W. M., "On Preventing Programming Languages From Interfering with Programming", IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975.

Melton, R. A., "Automatically Translating FORTRAN to IFTRAN", Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Military Standard, "Technical Reviews and Audits for Systems, Equipment and Computer Programs", Mil-Std-1521 (USAF), September 1972.

Millbrandt, W. W., and Rodriguez-Rosell, "An Interactive Software Engineering Tool for Memory Management and User Program Evaluation", Proceedings of the 1974 National Computer Conference 1974.

Miller, E. F., Jr., "IFTRANX-Exportable FORTRAN Extension for Structured Programming", General Research Corporation, Program Validation Research Project.

Miller, E. F., Jr., "Program Validation: The State-of-the-Art, General Research Corporation, Santa Barbara, California, August 1972.

Miller, E. F., Jr., "Extensions to FORTRAN and Structured Programming - An Experiment", General Research Corporation, RM-1608, March 1972.

Miller, E. F., Jr., "Technology for Automated Verification Systems", General Research Corporation, Paper for Aeronautical Systems Software Workshop, 1974.

Miller, E. F., Jr., et.al., "Structurally Based Automatic Program Testing", EASCON '74, Washington, D.C., October 1974.

Miller, E. F., Jr., "Toward Automated Software Testing: Problems and Payoffs", Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Miller, E. F., Jr., "RXVP: An Automated Verification System for FORTRAN", Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Miller, E. F., Jr., Melton, R. A., "Automated Generation of Test Case Datasets", International Conference on Reliable Software, April 1975.

Miller, E. F., Jr., "Experience with RXVP in Verification and Validation", Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Mills, H. D., "Chief Programmer Teams - Principles and Procedures", Report No. FSC 71-5108, IBM Federal Systems Division, 1971.

Mills, H. D., "Top-Down Programming in Large Systems", Debugging Techniques in Large Systems, Courant Computer Science Symposium 1, NYU (Editor, R. Rustin), 1971.

Mills, H. D., "Mathematical Foundations for Structured Programming", FSC 72-6012, February 1972.

Mills, H. D., "Reading Programs as a Managerial Activity", Working Paper, March 1972.

Mills, H. D., "On the Development of Large Reliable Programs", May 1973.

Mills, H. D., "The Complexity of Programs", Program Test Methods, Prentice Hall, 1973.

Mills, H. D., "The New Math of Computer Programming", Communications of the ACM, January 1975.

Mills, H. D., "How to Write Correct Programs and Know It", International Conference on Reliable Software, April 1975.

Mittwede, W. C., and Choate, K. P., "Operating System Validation Testing", Comtre Corporation, AD 724 717, January 1971.

Morgan, H. L., "Spelling Correction on Systems Programs", Communications of the ACM, February 1970.

Morris, J. B., "Programming by Semantic Refinement", SIGPLAN Notices, Vol. 8, No. 9, September 1973.

Mulock, R. B., "A Study of Software Reliability at the Stanford Linear Accelerator Center", Stanford University, August 1970.

Mulock, R. B., "Software Reliability Engineering", Proceedings of the Annual Reliability and Maintainability Symposium, January 1972.

Miyamoto, I., "Software Reliability in On-Line Real Time Environment", International Conference on Reliable Software, April 1975.

Moranda, P. B., "Status Report on Software Reliability Study for 1971", IRAL · MDAC-West-02-I07.

Moranda, P. B., and Jelinski, Z., "Software Reliability Predictions", Paper submitted to Symposium on Software Reliability, April 1975.

Moranda, P. B., "Predictions of Software Reliability During Debugging", 1975 Proceedings of the Annual Reliability and Maintainability Symposium, January 1975, Washington, D. C.

Moranda, P. B.,"Estimation of a Priori Software Reliability", Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Moulin, M., "Utilization du Systeme de Test et D'evaluation de Programmes (STEP) Pour la Mise un Point des Programmes", Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Myers, G. J., "Composite Design: The Design of Modular Programs", TR00.2406, IBM Systems Development Division, Poughkeepsie, N.Y., January 1973.

Naftaly, S.M., Cohen, M.C., "Test Data Generators and Debugging Systems - Workable Quality Control," Part I and II Data Processing Digest, Vol. 18, Nos. 2 and 3, February - March 1972.

Nassi, I., and Shneiderman, B., "Flowchart Techniques for Structured Programming," SIGPLAN Notices, Vol. 8, No. 8, August 1973.

Naur, P., "Proof of Algorithms by General Snapshots," BIT, Vol. 6, 1966.

Naur, P., "An Experiment on Program Development," BIT, Vol. 12, 1972.

Neely, P.M., "On Program Control Structure," Proceedings of the 1973 ACM National Conference, 1973.

Ng, E.W., "Mathematical Software Testing Activities," Program Test Methods, Prentice Hall, 1973.

405. Ogdin, J.L., "Designing Reliable Software," Datamation, July 1972.

406. Ogdin, J.L., "Improving Software Reliability," Datamation, January 1973.

407. Oliver, P., "COBOL '74 - Contributions to Structured Programming," National Computer Conference, May 1974.

408. Orgass, R.J., "Some Results Concerning Proofs of Statements About Programs," Journal of Computer and Systems Sciences, Vol. 4, 1970.

409. Osterweil, L.J., and Fosdick, L.D., "Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation and Error Detection," Dept. of Computer Science, University of Colorado, September 1974.

410. Osterweil, L.J., Fosdick, L.D., Automated Input/Output Variable Classification as an Aid to Validation of FORTRAN Programs, Report #CU-CS-037-74, Dept. of Computer Science, University of Colorado, September 1974.

411. Osterweil, L.J., Clarke, L., Smith, D.W., "A FORTRAN System for Flexible Creation and Accessing of Data Bases," Report #CU-CS052-74, Department of Computer Science, University of Colorado, August 1974.

Paige, M.R., and Miller, E.F., "Ranking Priorities in Testing Computer Programs," Proceedings of Computer Systems Design Conference, Industrial and Scientific Conference Management, Inc., Chicago, Illinois, 1972.

Paige, M.R., and Miller, E.F., "Methodology for Software Validation - A Survey of the Literature," General Research Corporation RM-1549, March 1972.

Paige, M.R., and Balkovich, E.E., "On Testing Programs," IEEE Symposium on Computer Software Reliability, May 1973.

Park, D., "Fixpoint Induction and Proofs of Program Properties," Machine Intelligence 5, American Elsevier Publishing Company, 1970.

Parnas, D.L., "Information Distribution Aspects of Design Methodology," Technical Report, Dept. of Computer Science, Carnegie-Mellon University, February 1971.

Parnas, D.L., "A Technique for Software Module Specification with Example," Communications of ACM, Vol. 15, No. 5, May 1972.

Parnas, D.L., "Response to Detected Errors in Well-Structured Programs," Technical Report, Dept. of Computer Science, Carnegie-Mellon University, July 1972.

Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," Communications of ACM, December 1972.

Parnas, D.L., "Some Conclusions from an Experiment in Software Engineering Techniques," Proceedings of the FJCC, 1972.

Parnas, D.L., Siewiorek, D.P., "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems," Technical Report, Dept. of Computer Science, Carnegie-Mellon University, July 1972.

Parnas, D.L., "The Influence of Software Structure on Reliability," International Conference on Reliable Software, April 1975.

Peters, L., "Managing the Transition to Structured Programming," Datamation, May 1975.

Pomeroy, J.W., "A Guide to Programming Tools and Techniques," IBM Systems Journal, Vol. 11, No. 3, 1972.

Poole, P.C., and Waite, W.M., "Portability and Adaptability," Advanced Course on Software Engineering, New York: Springer-Verlag, 1973.

Poole, P.C., "Debugging and Testing," Advanced Course on Software Engineering, Springer-Verlag, New York, 1973.

Popek, G.J., Kline, C.S., "A Verifiable Protection System," International Conference on Reliable Software, April 1975.

Presser, L., "Structured Languages," (position paper), National Computer Conference, May 1975.

Proceedings "Report of the Seventh Annual Data and Configuration Management Workshop," Electronic Industries Association Engineering Department, November 1973.

Prokop, J.S., "On Proving the Correctness of Computer Programs," Program Test Methods, Prentice Hall, 1973.

RADC, "Programming Support Library Program Specifications," Structured Programming Series, RADC Report, RADC-TR-74-300, Vol. VI.

Ragland, L.C., "A Verified Program Verifier," Ph.D. Thesis, University of Texas at Austin, June 1973.

Ramamoorthy, C.V., "Discrete Systems Representation and Analysis by Generating Functions of Abstract Graphs," IFIP Congress Symposium, New York, May 1965.

Ramamoorthy, C.V., and Chandy, K.M., "Optimization of Memory Hierarchies in Multiprogrammed Systems," Journal of the ACM, Vol. 17, No. 3, July 1970.

Ramamoorthy, C.V., Meeker, R.E., Sr., Turner, J., "Design and Construction of an Automated Software Evaluation System," IEEE Symposium on Computer Software Reliability, May 1973.

Ramamoorthy, C.V., and Ho, S.B.F., "Testing Large Software with Automated Software Evaluation Systems," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975, and International Conference on Reliable Software, April 1975.

Randell, B., "System Structure for Software Fault Tolerance," International Conference on Reliable Software, April 1975.

Rault, J.C., "Design Verification Techniques - A Review," International Conference on Reliable Software, April 1975.

Rault, J.C., "Extension of Hardware Fault Detection Models to the Verification of Software," Program Test Methods, Prentice Hall, 1973.

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

Reifer, D.J., "Automated Aids for Reliable Software," International Conference on Reliable Software, April 1975.

Rizza, J., and Hacker, D., "Quality Assurance Inspection and Test Tools - An Application," Workshop on Currently Available Program Testing Tools, Technology and Experience, April 1975.

Robinson, L., "Computer Systems Performance Evaluation (and Bibliography), IBM, November 1972.

Robinson, L., Levitt, K.N., Neumann, P.G., Saxena, A.R., "On Attaining Reliable Software for a Secure Operating System," International Conference on Reliable Software, April 1975.

Rose, C.W., "LOGOS and the Software Engineer," Proceedings of the AFIPS 1972 FJCC, 1972.

Ross, D.T., Goodenough, J.B., and Irvine, C.A., "Software Engineering: Process, Principles and Goals," Computer, May 1975.

Rowe, L.A., Hopwood, M.D., Farber, D.J., "Software Methods for Achieving Fail-Soft Behavior in the Distributed Computing System," IEEE Symposium on Computer Software Reliability, May 1973.

Royce, W.W., "Software Requirements Analysis: Sizing and Costing," Practical Strategies for Developing Large Software Systems, Addison Wesley, 1975.

Rubey, R.J., and Dulac, B., "Software Tools for Certifying Operational Flight Programs," Logicon, Inc., 1972.

Rubey, R.J., "New Approaches for Software Validation," Naecon 72 Record, 1972.

Rubey, R.J., "Quantitative Aspects of Software Validation," International Conference on Reliable Software, April 1975.

Rustin, R., ed., Debugging Techniques in Large Systems, Prentice-Hall, 1971.

Ryder, B.G., "The PFORT Verifier," Software Practice and Experience, Vol. 4, No. 4, October-December 1974.

Ryder, B.G., "The PFORT Verifier Users Guide," Computing Science Technical Report #12, Bell Laboratories, Murray Hill, New Jersey.

Sadowski, W.L., and Lozier, D.W., "A Unified Standards Approach to Algorithm Testing," Program Test Methods, Prentice-Hall, 1973.

Sande, G., "Program Execution Profiles," Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Saxena, A.R., Bredt, T.H., "A Structured Specification of a Hierarchical Operating System," International Conference on Reliable Software, April 1975.

Scherr, A.L., "Developing and Testing a Large Programming System, OS/360 Time Share Option," Program Test Methods, Prentice Hall, 1973.

Schick, G.J., and Wolverton, R.W., "Assessment of Software Reliability," Proceedings of German Operations Research Society, September 1972.

Schlender, P., "Application of Disciplined Software Testing," Debugging Techniques in Large Systems, Prentice-Hall, 1971.

Schmid, H.A., "On the Use of Interactive Programming Systems as a Tool for Structured Program Testing and Development," Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Schneidewind, N.F., "Analysis of Error Processes in Computer Software," International Conference on Reliable Software, April 1975.

Schwartz, J.T., "An Overview of Bugs," Debugging Techniques in Large Systems, Prentice Hall, 1971.

Seegmuller, G., "Definition of Systems," Software Engineering, NATO Science Committee, April 1970.

Severance, D.G., and Merten, A.G., "Performance Evaluation of File Organizations Through Modelling," Proceedings of the 1972 ACM National Conference, 1972.

Shneiderman, B., "Experimental Testing in Programming Languages, Stylistic Considerations and Design Techniques," National Computer Conference, May 1975.

Shooman, M.L., "Probability Models for Software Reliability Prediction," Statistical Computer Performance Evaluation, Academic Press, 1972.

Shooman, M.L., "An Introduction to Software Reliability," IEEE Symposium on Computer Software Reliability, May 1973.

Shooman, M.L., "Operational Testing and Software Reliability Estimation During Program Development," IEEE Symposium on Computer Software Reliability, May 1973.

Shooman, M.L., "Software Reliability: Measurement and Models," Annual Reliability and Maintainability Symposium, Washington, D.C., 1975.

Shooman, M.L., Bolsky, M.I., "Types, Distribution and Test and Correction Times for Programming Errors," International Conference on Reliable Software, April 1975.

Sintzoff, M., "Calculating Profiles of Programs by Valuations on Specific Models," Proceedings of the Conference on Proving Assertions About Programs, January 1972.

Sintzoff, M., Van Lansweerde, A., "Constructing Correct and Efficient Concurrent Programs," International Conference on Reliable Software, April 1975.

Sites, R., "Clean Termination of Computer Programs," Ph.D. dissertation, Stanford University, Stanford, California, June 1974.

Sloane, N.J.A., "On Finding the Paths Through A Network," Bell System Technical Journal, Vol. 51, February 1972.

Smith, R.W., "Measurement of Segment Relationship Execution Frequency," TRW System (#72-4912.30-31), March 1972.

Snowden, R.A., "Systems for the Preparation and Validation of Structured Programs," Program Test Methods, Prentice Hall, 1973.

Standard Data Corporation, "SYMBUG, Integrated Symbolic Debugging System User Guide," 1974.

Standick, T.A., "Extensibility in Programming Language Design," (position paper), National Computer Conference, May 1975.

Steele, S.A., "Experience with Software Testing Tools for Real Time Sensor Control System," Workshop on Currently Available Test Tools: Technology and Experience, April 1975.

Stevens, W.P., Meyers, G.J., and Constantine, L.L., "Structured Design," IBM Systems Journal, Vol. 13, No. 2, 1974.

Stillman, R.B., "FORTRAN Analysis by Simple Transforms," Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Strachey, C., "The Interaction of Software Engineering and Machine Structure," The Fourth Generation Infotech, Ltd., 1971.

Stucki, L.G., "A Prototype Automatic Program Testing Tool," FJCC, December 1972.

Stucki, L.G., "Automatic Generation of Self-Metric Software," IEEE Symposium on Computer Software Reliability, May 1973.

Stucki, L.G., and Svegel, N.P., "Software Automated Verification System Study," MDAC-W Report MDC-G5103, January 1974.

Stucki, L.G., "Automated Tools and Techniques Assisting in Software Development, A Pragmatic Approach to Software Reliability," MDAC-W Paper, April 1974.

Stucki, L.G., "Tools - Lessons Learned - New Strategies," Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Stucki, L.G., "Statistical Approaches for Programmers to Application Software Verification," Computer Science and Statistics: 8th Annual Symposium on the Interface, February 1975.

Stucki, L., Foshee, G., "New Assertion Concepts for Self-Metric Software Validation," International Conference on Reliable Software, April 1975.

Sudakow, R., "Software Reliability, The Development Cycle," Logicon, October 1974.

Sullivan, J.E., "Measuring the Complexity of Computer Software," Mitre Corporation Report No. MTR 2648, Vol. V, November 1973.

Sullivan, J.E., "Extending PL/I for Structured Programming," Mitre Corporation Report No. MTR 2353, March 1972.

Supnik, R.M., "Debugging Under Simulation," Courant Computer Science Symposium 1, June 1970; Debugging Techniques in Large Systems, Prentice-Hall, 1971.

Suzuki, N., "Verifying Programs by Algebraic and Logical Reduction," International Conference on Reliable Software, April 1975.

Tatman, J.C., "Achieving Proper Program Documentation," Journal of System Management, Vol. 22, No. 11, November 1971.

Taylor, S.M., "Loops in Computer Programs," Biological Computer Laboratory Report 4.3, University of Illinois, October 1970.

Teitelman, W., "Automated Programming - The Programmer's Assistant," Proceedings AFIPS 1972 FJCC, 1972.

Tenny, T., "Structured Programming in FORTRAN," Datamation, July 1974.

Thayer, T., "Understanding Software Through Empirical Reliability Analysis," National Computer Conference, May 1975.

Topor, R.W., "Interactive Program Verification Using Virtual Programs," Ph.D. Dissertation, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, December 1973.

Trauboth, H., "Guidelines for Documentation of the Scientific Software Systems," IEEE Symposium on Computer Software Reliability, May 1973.

Trivedi, A.K., Shooman, M.L., "A Many State Markow Model for the Estimation and Prediction of Computer Software Performance Parameters," International Conference on Reliable Software, April 1975.

Univac 90/70, "Program Test System," Hardware and Software Facts and Figures, Sperry Rand, 1973.

Van Noot, T.J., "System Testing - Taboo Subject," Datamation, December 15, 1971.

Ver Hoef, E.W., "Automatic Program Segmentation Based on Boolean Connectivity," Proceedings of the SJCC, 1971.

Von Henke, F.W., and Luckham, D.C., "A Methodology for Verifying Programs," International Conference on Reliable Software, April 1975.

Vyssotsky, V.A., "Common Sense in Designing Testable Software," Program Test Methods, Prentice Hall, 1972.

Vyssotsky, V.A., "Large Scale Reliable Software: Recent Experience at Bell Labs," International Conference on Reliable Software, April 1975.

Wagner, R.A., "Order-n Correction for Regular Languages," Communications of the ACM, Vol. 17, No. 5, May 1974.

Wagoner, W.L., "The Final Report on a Software Reliability Measurement Study," Report #TOR-0074 (4112)-1, Aerospace Corporation, El Segundo, California, 1973.

Waldbaum, G., "Evaluating Computing System Changes by Means of Regression Models," 1st Annual SIGME Symposium on Measurements and Evaluation, 1973.

Walker, A.W., "An Interactive Graphical Debugging System," AD 728 711, Naval Postgraduate School, June 1971.

Walter, K.G., Schaen, S.I., Ogden, W.F., Rounds, W.C., Shumway, D.G., Schaeffer, D.D., Biba, K.J., Bradshaw, F.T., Ames, S.R., and Gilligan, J.M., "Structured Specification of a Security Kernel," International Conference on Reliable Software, April 1975.

Walters, J.A., "Computer Aided Test Systems," Bendix Corporation, BDX 613 275, December 1970.

Wasserman, A.I., "Issues in Programming Design - An Overview," (position paper), National Computer Conference, May 1975.

Wegbreit, B., "Multiple Evaluations in a Extensible Programming System," Proceedings of the AFIPS 1972 FJCC, 1972.

Wegbreit, B., "The Synthesis of Loop Predicates," Communications of the ACM, Vol. 16, No. 2, February 1974.

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

Weinberg, G.M., "The Psychology of Improved Programming Performance," Datamation, November 1972.

Weissman, L., and Stacey, G.M., "An Interface System for Improving Reliability of Software Systems," IEEE Symposium on Computer Software Reliability, May 1973.

Wheeler, D.J., "The Limits of Complexity of Computer Systems," Proceedings of IFIP Congress 71, Amsterdam: North Holland, 1972.

White, J.R., and Presser, L., "A Tool for Enforcing System Structure," Proceedings of the 1973 ACM National Conference, 1973.

Whitten, D.E., and deMaine, P.A.D., "A Machine and Configuration Independent FORTRAN: Portable FORTRAN (PFORTRAN)," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975.

Williams, R.D., "Managing the Development of Reliable Software," International Conference on Reliable Software, April 1975.

Williamson, O.L., Dorris, G.G., Rybert, A.J., and Straight, W.E., "A Software Reliability Program," Federal Electric Corporation, 1970.

Wirth, N., "Program Development by Stepwise Refinement," Communications of the ACM, Vol. 14, No. 4, April 1971.

Wirth, N., "An Assessment of the Programming Language Pascal," International Conference on Reliable Software, April 1975.

Wolverton, R.W., "The Cost of Developing Large Scale Software," Practical Strategies for Developing Large Software Systems, Addison-Wesley, 1975.

Wong, P.J., "Application of Decision Theory to the Testing of Large Systems," IEEE Transactions on Aerospace and Electronic Systems, March 1971.

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

Wulf, W.A., "Programming Without the GOTO," Information Processing 71, North Holland Publishing Company, Software 1972.

Wulf, W.A., "A Case Against the GOTO," Proceedings of the ACM National Conference, 1972.

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

Wulf, W.A., "Reliable Hardware - Software Architecture," International Conference on Reliable Software, April 1975.

Yau, S.S., and Cheung, R.C., "Design of Self-Checking Software," International Conference on Reliable Software, April 1975.

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

Youngberg, E.P., "A Software Testing Control System," Program Test Methods, Prentice Hall, 1973.

Yourdon, E., "Making the Move to Structured Programming," Datamation, June 1975.

Yourdon, E., "Symposium on Structured Programming in COBOL," Datamation, June 1975.

Zahn, C.T., Jr., "Structured Control in Programming Languages," (position paper), National Computer Conference, May 1975.