RADC-TR-78-4, Vol I (of two)
Final Technical Report
January 1978

SOFTWARE MODELING STUDIES
Summary of Technical Progress

M. L. Shooman
H. Ruston

Polytechnic Institute of New York

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-78-4, Vol I (of two) has been reviewed and is approved for publication.

APPROVED: Alan N. Sukert

ALAN N. SUKERT
Project Engineer

APPROVED: Wendall C. Bauman

WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-78-4~~[illegible]~~-VOL-1	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: SOFTWARE MODELING STUDIES. Volume I. Summary of Technical Progress.	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report. 1 April 74 - 1 October 77
6. PERFORMING ORG. REPORT NUMBER: ~~SRS-112~~ Poly-EE77-042-VOL-1 SRS-112-VOL-1	
7. AUTHOR(s): M. L. Shooman, H. Ruston; Martin L. Shooman, Henry Ruston	8. CONTRACT OR GRANT NUMBER(s): F30602-74-C-0294
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Polytechnic Institute of New York, 111 Jay Street, Brooklyn NY 11201	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: P.E. 62728F; J.O. 5559~~[illegible]~~ 08
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441	12. REPORT DATE: January 1978
	13. NUMBER OF PAGES: 42
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Same (12 50p.)	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

DDC RECEIVED APR 4 1978

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

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Alan N. Sukert (ISIS)

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

Software Reliability Models	Markov Models
Software Availability Models	Bug Seeding/Tagging
Software Test Models	Statistical Test Models
Software Complexity Models	Natural Language Theory

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

This report documents research performed under RADC contract ~~F30602-74-C-0294~~ by Polytechnic Institute of New York in the area of software modeling. Research in the areas of software error, reliability, and availability models (such as Markov availability models, bug tagging estimates of initial error counts and error models incorporating error generation), test models and techniques (such as determination of the number of the tests necessary to execute all program paths and statistical test models), and complexity models (such as [illegible]nent measures like testedness and natural language theory

over

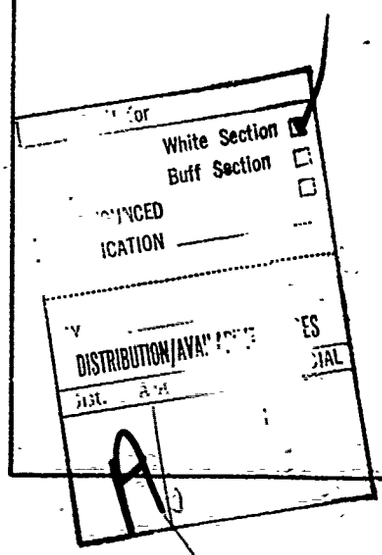
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

408 717

measures) that was described in previous progress and technical reports is summarized and unfinished research (in areas such as acceptance test models and automatic programming techniques) not previously reported is described. The significant results are highlighted along with their interrelations and potential.



FINAL REPORT

TABLE OF CONTENTS

SOFTWARE MODELING STUDIES

	Page
1.0 Summary and Results	
1.1 Objective	1
1.2 Organization	1
1.3 Major Results	1
1.4 Dissemination of Results	2
1.5 Major Accomplishments	2
2.0 Introduction and Overview	
2.1 State of the Art	4
2.2 Need for Modeling Research	4
3.0 Applications to Software Engineering	
3.1 Design Phase	6
3.2 Testing and Debugging Phase	6
3.3 Operational Phase	7
4.0 Software Error, Reliability, and Availability Models	
4.1 Introduction	8
4.2 Reliability Models	8
4.2.1 Macro Models	9
4.2.2 Macro Model with Error Generation	9
4.2.3 Micro Model	9
4.3 Techniques for Estimation of Error Counts	9
4.4 Markov Availability Models	10

		Page
5.0	Test Models and Techniques	
5.1	Introduction	11
5.2	Small Scale Testing	11
5.3	Data Selection Studies	12
5.4	Determination of the Number of Tests	12
5.5	Automatic Test Drivers	13
5.6	Statistical Test Models	13
5.7	The Exhaustive Test Model	14
6.0	Complexity Models	
6.1	Introduction	15
6.2	Application of Recursive Function Theory to Program Complexity	16
6.3	Component Measures for Evaluation of Software Complexity	16
6.4	Language Theory Measures of Complexity	17
7.0	Other Research in Progress	
7.1	Introduction	18
7.2	Acceptance Test Models	18
7.3	Comparison and Test of Software Reliability Models	19
7.4	Application of Graph Theory to Software Reliability	19
7.5	Correlation of Program Errors with Complexity Measures	21
7.6	Choice of Strategy in Software Revision	22
7.7	Effect of Organizational Structure on Software Development	23
7.8	Collection, Storage, and Retrieval of Software Reliability Data	24

	Page
7.9 Classification and Enumeration of Program Errors	24
7.10 Modifications of the PLAGO Interpreter	25
7.11 Small Scale Tests	26
7.12 Automatic Programming Techniques	27
8.0 References	30
9.0 Professional Activities	
9.1 Papers	33
9.2 Reports	35
9.3 Symposia	36
9.4 Talks and Seminars	37
9.5 Books and Notes	39
9.6 Technical Committees	39
9.7 Theses	40
9.8 Professionnel Awards	41
10.0 Personnel	42

EVALUATION

The necessity for producing more reliable, lower cost software for such military applications as command and control and avionics has led to the disire to develop newer tools, techniques, and predictive aids for improving the method in which software is currently being developed and tested. This desire has been expressed in such documents as the Joint Logistics Commanders Software Reliability Working Group report (Nov 1975) and in numerous industry and Government sponsored conferences and symposia. As a result, efforts have begun to develop and test mathematical models for predicting the error content of a software package as well as determining test criteria and measures of complexity. However, early efforts to develop such models have been fragmented and have not produced models with the desired accuracy and usability.

This effort was initiated in response to this need for developing accurate software predictive models and fits into the goals of RADC TPO No.R5A, Software Cost Reduction (formerly RADC TPO No. 11, Software Sciences Technology), in particular the area of Software Quality (Software Modeling). The report summarizes models developed in the areas of software reliability and availability models, software test criteria, and measures of software complexity. The importance of the model development mentioned in this report is that it represents the first cohesive attempt to solve the entire range of software modeling problems, and in addition to develop models that truly reflect the actual software development process, and thus provide more accurate, more usable models.

The models developed under this effort will provide much needed tools for use by software managers in adequately tracking the progress of a software development in terms of the success of testing. In addition, the models developed under this effort will provide the basis for further model development that will eventually produce a complete modeling capability for the Air Force in the software area. Finally, models developed under this effort will be applicable to current Air Force software development projects and thus help to produce the high quality, low cost software needed for today's systems.

Alan N. Sukert

ALAN N. SUKERT
Project Engineer

1.0 Summary and Results

1.1 Objective

This report documents the research performed under Contract No. F-30602-74-C-0294 by the Polytechnic Institute of New York for Rome Air Development Center during the period April 1, 1974 to October 31, 1977. Research described in previous progress and technical reports is summarized and unfinished research not previously reported is described. The significant results are highlighted along with their interrelations and potential.

1.2 Organization

The technical contents are organized into four major subdivisions, Chapters 4.0, 5.0, 6.0, and 7.0. Chapter 1.0 and the following two chapters provide summary, introduction, and applications of the research performed.

Chapter 4.0 contains a technical summary of the work in Reliability Modeling. Similarly, test models and techniques are described in Chapter 5.0, and complexity models in Chapter 6.0. In Chapter 7.0, newly started or continuing projects are introduced.

Chapter 8.0 contains the relevant references for this document. Chapter 9.0 contains the complete list of papers, reports, talks, symposia, books, and theses. Chapter 10.0 lists the researchers supported in part or in whole by this contract.

1.3 Major Results

The major thrust of the work is divisible into three areas: software reliability and availability models, test models, and complexity models.

Building upon previous modeling work by Shooman in the reliability area, new macro and micro reliability models have been developed, namely:

(1) a Markov availability model;

(2) estimator formulas and tagging methodology for estimating the initial and subsequent number of errors;

(3) the macro model was made more realistic through the incorporation of error generation effects;

(4) a micro model based upon path traversal of the software.

The approach in the area of testing has been to classify and model as well as to provide test techniques. The research has resulted in:

(1) a new method of test data selection;

(2) a classification of types of tests;

(3) the construction of automatic test drivers;

(4) a statistical model of testing;

(5) a model of an exhaustive test.

Complexity measures have been sought which relate problem complexity to programming effort and reliability. The major accomplishments were:

- (1) modest advances in the application of the classical theory of recursive function theory to software problems (the method has practical drawbacks).

(2) development of software structural measures such as accessibility and testability;

(3) experimental verification and theoretical proofs that Zipf's law applies to computer languages;

(4) development of estimator formulas for computing the Zipf's law complexity measure.

1.4 Dissemination of Results

An important aspect of any research is the prompt dissemination of results so that (1) other researchers may explore the validity of the conclusions, and (2) practitioners may assimilate the techniques to their work. A complete list of all publications, either fully or partially supported, correlated to this contract appears in Chapter 9.0. There were 20 journal and conference papers, 22 technical and progress reports, 31 talks and seminars, and 5 Master's, Engineer, and Doctoral theses.

1.5 Major Accomplishments

The following major accomplishments by senior investigators deserve special mention:

- Two text books are nearing completion (M. Shooman, "Software Engineering: Reliability, Design, Management," Polytechnic Institute of New York, 1977, and H. Ruston, "Programming with PL/I," McGraw-Hill Book Company, New York, 1978).

- Two of the published papers won prizes as best technical paper of the conference (M. Shooman and S. Sinkar, "Generation of Reliability and Safety Data by Analysis of Expert Opinions," Proceedings of 1977 Annual Reliability and Maintainability Symposium, Philadelphia, Pa., and M. Shooman and A.Laemmel, "Statistical Theory of Computer Programs--Information Content and Complexity," Digest of Technical Papers, IEEE COMPCON 1977, pp 341-347.

- The leadership of two international symposia on Software Engineering included the senior investigators on this contract. These conferences were: 1975 International Conference on Reliable Software, Los Angeles, April 21-23, 1975, and Polytechnic Symposium on Computer Software Engineering, New York, April 20-23, 1976.

- Two new courses incorporating selected research in this area have evolved. These are: a graduate course CS606 - Software Engineering, developed by Professors Shooman, Laemmel, and Flynn, and an undergraduate course CS204 on Data Structures, developed by Professor Ruston.

2.0 Introduction and Overview

2.1 State of the Art

During the course of this contract there has been a dramatic change in the developing field of Software Engineering. Between 1968 and 1972 the military services were the first to point out the tremendous costs and difficulties associated with writing software. The growth of software complexity and its concommitant costs were spurred by increasingly more complex mission requirements and the vast array of new powerful computer hardware.

The first national symposium in the area of software reliability was held in 1973, and the first professional organization devoted to this area was the IEEE Computer Society Technical Committee on Software Engineering, formed in 1974. Subsequent conferences have been heavily attended, thus demonstrating the high interest in the field of software engineering.

Many of the sessions at these conferences focus on the need for qualitative and quantitative theory, design tools, and development techniques. In the past, progress has been slow, but lately it has been accelerating. Two common conference themes are modern versus classical program design, and probabilistic models versus program proofs. We still lack well-tested tools and techniques for the engineering of software, and those which exist are just beginning to be applied. Continued effort is necessary as computers increasingly pervade all phases of our life.

As a further complication, an entirely new class of software problems will occur during the late 1970's and early 1980's due to the microprocessor. Now that every electrical engineer has a powerful CPU and 500 to 2000 words of Random Access and Read Only Memories available at low cost, there will be a tremendous growth in the number of dedicated processors and their associated programs. Thus, the question of whether to use distributed processing is vacuous, and distributed microprocessors will proliferate electronic circuitry unless the military legislates against its use except in CPU's. One of the unique features which will accompany the introduction of the microprocessors will be volume production of both hardware and possibly the software, in both civilian and military applications.

2.2 Need for Modeling Research

The software industry needs a wide variety of design, analysis, and management tools to develop software in an efficient and cost effective manner. In some cases merely the classification and codification of design and testing approaches represents an advance; however, our end goal is analysis and design techniques. The specific needs are outlined below:

In the reliability area we specifically need:

- to estimate the dollar and man-hour costs of removing software errors,

- to rationally decide whether to release, continue testing, or completely redesign a software product,
- to analyze the initial and life cycle costs of competitive designs,
- to provide definitions of terms and techniques for the collection of software reliability data,
- to provide quantitative goodness measures for assessing a particular design or a new technique.

In the testing area, the specific needs are:

- to obtain parameter values for some of the reliability and error models,
- to classify and model the testing process,
- to suggest new means and strategies of testing,
- to estimate the costs of software testing,
- to measure and manage progress during the test phase of software development.

The complexity of the posed problem, the design developed, and the code produced, interrelate in many ways. The problems center about how to define the relevant relationships, model the effects, and obtain practical complexity measures related to:

- development time and costs,
- amount of testing and number of bugs,
- type of architecture and speed of computer,
- memory size and type,
- skill and number or programmers,
- maintenance and life-cycle costs.

The above gives an overview of the research needs which are addressed in Chapters 4 through 7.

3.0 Applications to Software Engineering

3.1 Design Phase

Design is inherently an iterative process, involving many separate steps, repeated at different stages of the project. During the initial stage the proposed design and its alternatives are evaluated for feasibility and performance. One of the first steps is to gauge the overall scope and feasibility of the job by assessing its complexity. The common approach is to equate it with similar past systems, on which data exists, and draw comparisons. Such a technique is imprecise and of diminished usefulness in a military environment, where most new tasks eclipse their predecessors in complexity.

Some of the complexity measures developed under this contract (c.f. Sec. 6.4) are related to fundamental properties of the problem and should allow the extrapolation of past results into the future. To utilize the complexity model during early design, we tabulate the input and output quantities, and estimate what proportion of the available operators in the chosen language will be used. These values are substituted into a formula which predicts the operator-operand (token) length. One then relies on historical data for the ratios of errors/token and man-hours/token. Although these ratios depend on historical data, the scaling procedure based upon token complexity is objective, repeatable, and has been shown (in our small number of retrospective estimates) to be reasonably accurate.

The reliability model discussed in Sec. 4.3 can be used to roughly predict the software failure rate and mean-time between software errors at the design stage. One of the inputs to this model is the instruction length of the program. This can be estimated from the token length and historical data on the number of instructions/token in the chosen language. An additional parameter needed in the model is the error discovery rate constant. This constant relates the number of occurrences of external operational errors to the number of residual internal errors. (Note: An internal error is a potential one which only becomes an actual external error when the particular data which excites it is processed.) Much of the research described in this report involves techniques for measuring these model parameters and testing the ease and accuracy of application of the above-mentioned formulas (c.f. 4.2, 5.2, 7.3, 7.5, 7.11).

3.2 Testing and Debugging Phase

The contributions to this phase contain several techniques for: selecting test data, calculating the number of tests, and automatic testing. In addition, contributions were made to classifying software by the ease of testing and classifying tests by their degree of completeness.

The technique for selecting test data involves the mapping of constraints on the output into the equivalent set of constraints on the input test data. A description of this technique is given in Section 6.3.

The knowledge of the number of tests is essential for bounding the test costs and developing a test plan. The method described in Section 5.4 consists of a sequence of stages. First, the control structure of the software is modeled by a graph. Second, matrices representing the graph structure are constructed. Lastly, matrix manipulations are performed to obtain the desired number of tests.

The automatic testing technique developed in this contract yields a test driver. This driver analyzes the source code and modifies it. The modification allows the generation of test data which exercises the software in a specific manner. The modification is such that each IF statement is associated with either a bit string variable of 0 or 1, representing the two outcomes. Similarly, the passage of a DO WHILE loop is controlled. Lastly, various program modules are exercised for the initial values. The details of this method are given in Section 5.5.

The classification of software is based upon accessibility of certain modules, ease of testing such modules, and the use of resources in such tests. The details of this classification are given in Section 6.3.

3.3 Operational Phase

Many of the models described previously can also be applied to the operational phase. As an example, the release of large software systems to the field is accompanied by a flurry of software errors. The field debugging process can be controlled by the reliability models described in Section 4.3.

Once steady-state prevails in the field, the most significant parameter of software performance is its availability. The Markov availability models described in Section 4.1 may be used to predict the steady-state availability of the software.

The immediate predecessor of the operational phase is the acceptance test. This test delineates the transition between development testing and operational deployment. Section 7.2 describes one type of such test. The philosophy of the acceptance test divides the acceptance into two parts. The first part involves testing of each feature within the nominal range. The second part involves testing of stressful conditions as well as boundaries of the range. The software is accepted only if a prestated high percentage of the tests are successful.

Section 7.2 delineates the suggested procedures to be followed if the software is not accepted.

4.0 Software Error, Reliability, and Availability Models

4.1 Introduction

We shall present here brief summaries of work documented in technical reports and papers relating to models of software errors and their effect on operational reliability and availability. These models are needed for a number of technical and management functions, namely:

- To estimate the cost and development time related to the excision of the errors initially residing in the program.
- To provide an analysis technique for one of the three decisions; that is, (1) to continue more testing, (2) to terminate testing and to accept the software, and (3) to abort a hopeless effort.
- To provide a technique for cost-reliability trade-off among competitive designs. Such trade-off calculations are needed, for example, to choose between a time consuming technique which produces software with few errors and a fast technique leading to many errors.
- To provide definitions and techniques for operational data collection and measurement. The collected data is needed for the determination of model parameters and the measurement of operational system reliability.
- To provide quantitative measures of goodness which enable researchers to assess the promise of a new technique. This can be done by contrasting the error content and reliability of software constructed with and without such a technique.

The research supported under this contract has advanced the field by providing:

- a Markov model of availability,
- an estimate of initial error content of software based upon a tagging procedure,
- a software reliability model incorporating errors generated during the debugging process,
- a micro reliability model based upon path traversal of the software.

4.2 Reliability Models

The initial trust of modeling focused on: (1) the number of software errors, (2) the rate of their removal during the testing phase, and (3) the operational failure rate resulting from residual errors. This work leads to three phases of models, to be described here.

4.2.1 Macro Models

The model developed in the first phase portrayed just the gross features of interest. It provided expressions for the number of remaining errors as the debugging process unfolded. Certain reasonable assumptions were made to relate the number of residual errors to the operational system reliability. One of the assumptions postulated the constancy of the sum of the removed errors and the remaining errors.

The model of the first phase was named the macro model, because it only considered the gross or exterior behavior of the removal process. Such a model is analogous to modeling the exterior input-output relationships of the system without regard as to the interior structure of the system. The macro model is described in References 13, 14, and 15.

4.2.2 Macro Model with Error Generation

The macro model was refined in the second phase. The refinement consisted of incorporating the errors generated during the debugging. The generated errors were modeled in several ways. Their rate of removal was modeled as a function of the project's manpower, giving rise to a related investigation of manpower deployment strategies. The complete development of the error generation model and the effect of deployment strategies is described in Reference 7.

4.2.3 Micro Model

The completion of the second phase, just described in Section 4.2.2, indicated that further improvement in the fidelity of the model will necessitate the incorporation of factors related to the structure of the software. The model which was investigated is based upon the program flowcharts (or equivalent representation, e.g., pseudo-code), its paths, and the execution and the error frequencies along each path. Such a model is based on the assumption that the software system can be decomposed into a modest number of mostly independent paths (structured programs generally satisfy this assumption). Any gross failure of the decomposibility or independence features invalidate such a model. The micro model is described in Reference 16.

4.3 Techniques for Estimation of Error Counts

A method has been developed for the estimation of errors residing in software at any stage of its construction. This method is based on either seeding or tagging of errors in the program under construction. The initial idea of seeding comes from statistical techniques for estimating the size of wild life population, and was first adapted by H. Mills (Reference 17). Mills experimented with the addition of seeded bugs to those naturally residing in the program. After partial debugging by another programmer (i.e., one who does not know of the seeded bugs), the bugs found are separated into the seeded and the natural categories. By using the total number of seeded bugs, the number of seeded bugs found, and the number of natural bugs found, Mills obtained estimates for the initial number of natural bugs. A major problem in applying this technique is the difficulty in "manufacturing" realistic bugs for seeding.

To avoid this difficulty it has been suggested (Ref. 18), that instead of seeding program errors, a tagging technique be used. This technique is implemented by having the program debugged independently by two or more testers. A different person, acting as the analyst, analyzes the lists of bugs found and identifies common bugs found by two or more programmers. The number of common bugs found by multiple programmers serves the same purpose as the seeded bugs in the former techniques. The disadvantage of the method is the need for additional personnel (i.e., the independent debuggers). However, this cost is largely offset by the extra bugs that the additional personnel finds.

The tagging method has a number of advantages over the seeding one:

1. The difficulty of manufacturing realistic seeded bugs is eliminated.
2. None of the testing and debugging funds are used for the finding of known seeded bugs.

Because of the promise of the method, a major effort was undertaken resulting in a comprehensive report (Reference 8). This report derives several estimator formulas for mean and variance for different sample sizes. The report also describes how the method is to be used for the case of two, and for the case of several debuggers. A small scale experiment has been performed, and the data is presently being analyzed.

The method is especially useful in estimating the initial number of bugs, to provide measures (1) of the cost of testing and (2) of a parameter in the reliability model. It is also useful for obtaining information for management decisions at certain key points of software development.

4.4 Markov Availability Models

Models have been developed (Ref. 6) for the determination of availability of an operational (i.e., field deployed) computer software. For many systems the reliability is not the crucial measure, but rather the ratio of up-time to the total operating time, that is, the system availability.

The models developed are based upon Markov processes. The initial up-state assumes the system to be running with undiscovered bugs. When the first software error is discovered, the system enters the second up-state which represents n-1 bugs. The model thus consists of a sequence of up and down states. The transition probabilities between up and down states are essentially the software failure rates. The transition probabilities between down and up states are the probabilities of repair taking place in a given time interval.

The model leads to difference and differential equations for the state probability (e.g., the probability that the system is in a particular up or down state). The reports show solutions for several different cases involving various assumptions on the failure and repair rates.

5.0 Test Models and Techniques

5.1 Introduction

We shall present here brief summaries of work documented in technical reports and relating to test models and techniques. These models are needed for a number of technical and management functions, namely:

- o To obtain parameter values for certain models described in Chapter 4.
- o To classify and model the testing process.
- o To suggest new means and strategies of testing.
- o To estimate the costs of software testing.
- o To manage and measure progress during the test phase of software development.

The research supported has advanced the field by providing:

- o a new method of test data selection,
- o a classification of types of tests,
- o the construction of automatic test drivers,
- o a statistical model of testing,
- o a model of an exhaustive test.

5.2 Small Scale Testing

In order to explore the verification of theoretical models (e.g., the tagging estimates), and to estimate parameters needed for other models, four programs have been written by student programmers and careful records were kept on their debugging experience. The initial effort and the four programs are described in Ref. 5 (pp. 22-31), along with the reporting scheme.

The four programs were constructed in increasing order of complexity: (1) salary payroll adjustment, (2) roots of a cubic equation, (3) library search, and (4) ballot counting procedure. Each debugger recorded: (1) the number of bugs and their types discovered during each test run, (2) the time required for analysis of each error and the computer time of the run needed to correct the error, and (3) the chronology of removal of inherent bugs and bugs generated during the testing phase.

As the data reduction continues, we intend to utilize the new measure of complexity (i.e., ZipF's Law, see Ref. 9) to explore the relationship between complexity and the initial number of program bugs. We also intend to correlate the Zipf's Law measure of complexity with intuitive subjective estimates of relative complexity by programmers (e.g., consensus estimation, see Reference 19).

5.3 Data Selection Studies

The crucial problem in testing is the selection and generation of an appropriate set of test data. This difficult problem has been studied from several viewpoints. One such view will be described here; several other viewpoints appear in the sections to follow.

The method used in this study uses constraints imposed on the variables. Tracing of all constraints resulted in the determination of bounds on the variables, and thus construction of test input data. The method requires the construction of dependency tables and has been described in Reference 1 (pp. 10-12) and in Reference 2 (pp. 33-50). A more complete treatment is given in Reference 20. At present the method is very tedious because of a laborious hand-tracing through the dependency tables. Future efforts will attempt to mechanize the effort (e.g., by perhaps writing a program replacing the hand-tracing) and to apply it to modest size programs. Even though the present method is traceable to programs of any size, the tediousness of the hand-tracing makes it impractical for other than small programs.

5.4 Determination of the Number of Tests

One of the pertinent questions of interest is the number of tests required of software, so as to verify such software to a desired degree of completeness. A complete verification requires perhaps an exhaustive test, that is, a test for all possible sets of inputs. Obviously such a test is almost always impractically large. Consequently, practical considerations demand lesser levels of testing. Hierarchies of such tests have been categorized (Ref. 21, Chapter 4) and assigned level numbers[1] from 0 to 4. Type 0 test exercises each program statement. The highest level is an exhaustive test. The in-between levels represent tests more complete than the 0 type but not exhaustive (e.g., level 2 is execution of all paths in the program). Also intermediate levels can be introduced, such as type 1.5 for example, which identify all the levels of the one level and some of the second level.

We have investigated analytical methods for computing the number for type 2 tests. This work was reported in Reference 2 (pp. 3-11), Reference 3 (pp. 40-49), and Reference 4 (pp. 10, 14-16). A more detailed discussion is given in a technical report presently in typing. The method starts with a graph of the control flow of the program. Theorems from graph theory have been applied to determine the number of type 2 tests. This entails visual search through graphs, which is very cumbersome for large problems. The alternate approach is to represent the control graph by a connectivity matrix. Such a matrix can then be manipulated via matrix transformations to yield the desired number. The matrix method is amenable to computer processing.

5.5 Automatic Test Drivers

Another approach to the generation of test inputs evolved in the construction of a so-called test driver. Such a driver is a generalized program. The

[1] Two different scales for levels have been introduced. The earliest scale went from 0 to 3. The newer scale extends from 0 to 4.

input to the driver is the program to be tested. The driver generates signals necessary for forcing the traversal of each program path.

The initial driver was implemented in LISP and is described in Reference 5 (pp. 14-17). Further work is reported in References 12 and 22. These references describe the progress achieved in the development of the driver. In particular, the present implementation is in PL/I, which allows driver testing of wider classes of programs. The initial version, for example, was restricted to the loopless program; the present version no longer suffers this constraint.

The essence of the algorithm is to associate a binary digit with the predicate (i.e., condition) of each decider (implemented in PL/I by the IF statement). The state of all the deciders can then be represented by a binary number made up of an ordered concatenation of the individual digits, with each digit representing a single decider. The algorithm generates the entire valid set of such binary numbers, and derives the appropriate driving signals from them.

Similar techniques are used to reduce the testing of a DO loop to just the first and the last passes. Such a reduction greatly reduces the run time of a complete test. For the details, the reader is again referred to the cited references.

It should be observed that the assumptions on the control structure being composed of only simple sequences, repetitions, and selections, are satisfied by a structured program. With the present emphasis on usage of structured programming techniques, these assumptions are not very limiting.

5.6 Statistical Test Models

A statistical test model has been developed which relates different program errors to the input data set (or sets) which excite and thus display a particular error. The model also gives the probability that these errors will cause the program to fail. This work was reported in References 3 (pp. 19-30), and 4 (pp. 17-19). A more detailed description of the model appears in Reference 23.

The model assumes that there are N total possible test inputs, (i.e., input data sets). It is further assumed that each input is equally likely to occur either as the tester's or the user's input. A certain number of these inputs, say W, will result in operational errors. During the development of the program only the subset t of the N possible inputs is used to test the program. The model relates Pe, the probability of the error occurring during the program use, to the parameters W, N, and t. The model also yields a rectangular grid representing the test process. Each grid point (j,i) is interpreted as the excitation of the bug i by the input j. The graph also portrays the two situations when (1) a single input excites several bugs, and (2) different inputs excite the same bug. These two situations are displayed by either a horizontal or vertical interconnection of grid points.

A related study utilizes a similar model to evaluate the probability that a particular sequence excites a bug. This work is reported in Reference 4 (pp. 17-19).

5.7 The Exhaustive Test Model

It is obvious that an exhaustive test is prohibitive in time and cost for almost all practical programs. Nevertheless, the number of tests for an exhaustive test plan still represents a useful upper bound on the actual number of tests that will be used in practical testing. These upper bounds have use in comparing the test efforts needed to test different programs.

The knowledge of the number of tests needed for an exhaustive test is needed if one wishes to use test coverage (that is, the number of the tests performed to the number of possible tests), as a figure of merit. Because of these applications, the modeling of an exhaustive test was studied and the results are reported in References 12, 21, 22, and 24.

The exact computation of the number of tests was made in Reference 24, for a particular example of a small program. In this example, the program extracted the roots of a quadratic equation, and was designed in PDP-8 assembly language. The number of exhaustive tests depended upon the range of possible values for the three coefficients of the quadratic and the 12 bit word length of the computer.

6.0 Complexity Models

6.1 Introduction

The fundamental problem in the development of software is to construct a product of minimum cost which still meets the quality and reliablity design specifications. When the product is completed records often show that the development costs greatly exceed the initial projection. A major cause of the poor estimation is the strong influence of complexity and the difficulty in measuring and assessing its effects.

Complexity, to be more specific, has a major effect on:

- o development costs,
- o development time,
- o amount of testing,
- o number of bugs,
- o memory needs,
- o size of host computer,
- o skill and number of programmers,
- o maintenance,
- o life-cycle-costs,

and many others. Complexity is a very elusive type of measure, with its importance perceived by everyone but in an ambiguous and fuzzy fashion.

Complexity affects all the above listed developmental parameters in a nonlinear, hard-to-determine way. Clearly one needs a more sophisticated mesure than a mere count of the number of lines of code. There are many examples of simple large programs and complex small programs. The problem of complexity is a difficult one as is evidenced by the number of highly competent researchers working on this problem and the relatively few important results published so far.

The research supported under this contract has advanced the field by providing:

- oo A comprehensive study of classical methods of complexity with emphasis on recursive function theory. The results were theoretically appealing; however, practically difficult to apply and extend.

- oo An approach in which software complexity is decomposed into simpler software components. One such component is accessibility, which is a measure of the software structure. Another component is testability, which is the ease with which a module may be accessed and tested. A third component is testedness, which measures the frequency of execution during the testing phase. This approach shows promise but more work is needed.

- oo A measure of complexity based on Zipf's Law and other results in natural language theory. This approach links known results from natural language and information theories to software complexity

questions. [The paper on this subject (Ref. 25) has received the best paper award at the Fall 1977 IEEE COMPCON conference.] The results give answers to such questions as the expected length of program given the initial estimate of the number of variables and the types of needed algorithms. The formulas derived from fundamental principles give answers which correlate well with those obtained independently through software science formulas (Reference 27).

6.2 Application of Recursive Function Theory to Program Complexity

The initial study of software complexity focused on the theories of computational complexity, with the intent of extending such theories to realistic programming problems. Recursive function theory was selected as the most promising candidate for such a study.

Strictly speaking, recursive function theory applies solely to mathematical functions. However, the programming solution to a problem often can be viewed as a function or functional. Thus, the first question of interest is how to reduce a general program to a function. In theory, each output variable can be related to input variables, but the practical problem of obtaining the mathematical functional relationship is not an obvious one. This is further complicated in that classical recursive function theory deals with integer variables.

In our work we extended recursive functions to the domain of non-integers and character strings (Reference 10). Unfortunately, the method is difficult to apply and suffers from several limitations. Consequently, we conclude that the method is of limited use for classifying practical problems in terms of their complexity.

6.3 Component Measures for Evaluation of Software Complexity

One approach to measuring complexity is to decompose it into simpler, easier to comprehend, components. Such an approach has been undertaken and in fact, some of the more promising measures can also be applied to determining the quality of software.

The first component developed and studied was named accessibility. Accessibility measures how easily a module can be reached and thus describes the difficulty in testing such a module (Reference 3, pp. 53-55). The second component was named testability. This component used both accessibility as well as a measure of resources required to test a program module. (Reference 3, pp. 55-56). The third component was named testedness and was defined as a function of testability and the frequency of execution during the testing phase (Reference 3, pp. 56-57).

Additional work on these concepts is described in References 20 and 26. This work is very promising but more effort is needed on obtaining more components and correlating them with the overall software complexity.

6.4 Language Theory Measures of Complexity

The use of number of statements as a measure of complexity is often used. It is recognized that such a simplistic measure does not truly gauge the program. Often there are programs with a few lines considerably more complex than programs with many lines. As a better measure Halstead (Reference 27) suggested operator and operand count. The analogy between operators, operands and verb nouns suggested the application of Zipf's Law from natural languages to programming languages.

The relationships between number of operators and operand types, as well as their frequency of occurrence, was shown to follow the basic Zipf's Law. An extended form of Zipf's Law has been derived which more closely models some of the experimental data.

The theory culminates in an equation which relates operator and operand length to the number of types (References 9 and 25). If we estimate early in the design the number of input and output varibles and operators (which will be used in the program) we can obtain an estimate of the program length.

These results correlate well with the results which Halstead has obtained using software science techniques (Reference 21, Chapter 3, and Reference 27).

7.0 Other Research in Progress

7.1 Introduction

As is usual in any research endeavor, there are a number of tasks which are in different stages of completion. Some of these represent new work, not reported in previous progress summaries. Other work has been reported earlier, but has not as yet reached the point where a comprehensive technical report is justified.

The brief sections of this chapter describe such tasks. These descriptions have been included, to record the ideas and their stage of progress, in order to provide continuity in the follow-up research.

The eleven technical sections of this chapter can be broadly classified into the following categories:

1. Models: Sections 7.2, 7.3, 7.4
2. Complexity: 7.4, 7.5
3. Software Management Tools: 7.6, 7.7
4. Data Collection: 7.8, 7.9, 7.10, 7.11
5. Design Techniques: 7.12

Each of these sections is self-contained with appropriate literature references to related work. It is anticipated that several of these will be developed into comprehensive technical reports, while others will be incorporated in future research.

7.2 Acceptance Test Models

Most of our research on testing has involved the development phase of software production (Ref. 11, 12, 22, and 24). This section describes some preliminary work which we have done on the structure of an acceptance test. Such a test is one of the most important milestones in the management of software development. An acceptance test is basically a vehicle whereby the contractor convinces the contractee that the software is good, correct, and should be accepted. The test which we have proposed consists of two parts.

1. A group of k test cases are selected which test each feature or mode of operation of the program. All parameters are selected well within the normal range of operation.
2. A collection of n test cases are devised such that they include all known extreme and difficult cases which constitute n_1 tests; the remaining $(n-n_1)$ cases are distributed over the normal range of operation.

The specific test cases are devised by the contractee (or his representative if third party testing is used) and are unknown to the contractor. If all k cases run successfully, and at least x% of the n test cases, then the software is accepted; however, the contractor is given a short amount of time to fix the n(1-x/100) errors. If either part 1 fails or part 2 does not meet the required threshold of x% successful tests the contractor is given time to improve the software. After the software improvement the acceptance test (i.e., both parts) is rerun with different test cases.

Work is presently continuing on these ideas with regard to identifying the contractor and contractee risks and relating the test results to the software reliability.

7.3 Comparison and Test of Software Reliability Models

Major questions in the research and application of software reliability models concerns the basic assumption, range of applicability, and ease of use of one model versus another. A good summary of the models proposed in the past and their comparison is given in References 28 and 29.

Answers to the above questions can be developed in several ways. One obvious approach is the theoretical investigation of the models, their underlying assumptions, and their limitations using mathematical statistics. This approach, taken by several workers (c.f. Ref. 14), suffers from drawbacks. In some cases the computations are intractable. In other cases it is difficult to assess the practical impact of the nonexistence of an entity (e.g., a particular moment) or the bias of an estimator.

The best overall test of such models is to use them for the analysis of sufficient actual field data with well-known outcome. If the predictions by-and-large agree with the field experience, we will be less concerned with the mathematical quirks. However, we never have field data in sufficient varieties to relieve our concern.

In order to supplement the field data, we propose to generate additional data by simulation (Reference 30). We may fabricate data of particular type to pin-point and study certain model weaknesses. Suppose, for example, that a reliability model is suspected of being insensitive to changes in failure rate in time. We can generate three sets of simulated data: (1) with constant failure, (2) with linearly increasing failure, (3) with exponentially decreasing failure. We can now study how the model responds to these three pure data pattern. Such study is not possible with real data which contains many patterns.

7.4 Application of Graph Theory to Software Reliability

The application of graph theory to problems in software reliability was studied. The motivation for this study is the fact that linear graphs (i.e., vertices connected by edges in various configurations), suggest themselves as

logical structures for representing computer programs. However, the detailed application of these concepts is not obvious. To be sure, some elementary approaches suggest themselves at the outset, and the flowchart concept, so basic to computer program documentation, is very close to a linear graph. For example, the vertices of the graph may be operational symbols of the chart, and similarly, the graph edges are the flow lines of the chart. Alternatively, the vertices may represent stages reached in the program, and the edges indicate the passage between stages (see Ref. 21, pp. 3-73ff). Such models can be used to study the path structures in the code. This in turn can relate to the kind of testing required to achieve various levels of software reliability. Covering a graph with paths and related concepts form a standard part of graph theory. The interplay between this material and software testing algorithms has received attention from a number of investigators.

Graph theory may contribute to other aspects of software reliability, as well as forming the basic representational structure just described. Several such aspects have been considered. These are briefly described below for possible future use even though they were not carried beyond the initial stages. These aspects deal with complexity and fundamental characterization of software.

As in the case of hardware, complexity as a concept carries both positive and negative aspects for reliability. On the one hand complex systems may be expected to be less reliable than simple ones; on the other, a major technique for improving reliability is enhancing system design which may increase complexity. It becomes clear that a conceptual use of "complexity" is not satisfactory, but specific and quantifiable definitions are required. In graph theory there are several definitions of graph complexity, the most well-known being cyclomatic complexity (discussed in Ref. 21, pp. 3-73ff and Reference 32). Two other measures of complexity are well defined. One uses the information content in a graph and its underlying automorphic group structure and was developed largely by Monshowitz (see Reference 33). Another approach to complexity has been formulated by Minoli and is described in Reference 34. He specifies a set of desirable properties for a complexity measure and defines its mathematical form. These measures of complexity were not developed with software applications in mind. It is likely that more appropriate measures can be produced by starting from basic needs of software analysis.

As another approach to complexity, consider activity on the linear graph or network (representing computer software) rather than the static (logical) structure alone, as is the case for the complexity measures mentioned above. One such approach is due to Flynn and is discussed in detail in his dissertation (Reference 35).

Though most people see at once a connection between software code, flow charts, and linear graphs, deep consideration exposes a lack of fundamental characterization. One wishes to employ mathematics to the study of a variety of computer software problems, but in fact, it is not clear what kind of mathematical object represents a particular computer software. Indeed, the

representation may differ depending upon need and intended usage. For example, consider the characterization of complexity. How should one represent the software and then impose a goal oriented definition of complexity upon the resulting mathematical object? In any case, regardless of specific representations, one should have a mathematical characterization of the software. Study indicates that such characterization would be likely to follow set theoretic formulations. This in turn leads to linear graphs which have a direct correspondence to subset structures. This basic characterization, however, has not been completed as yet.

An aid to the study of the interplay between graphs and code is the automatic generation of linear graphs which represent computer code. A program to take FORTRAN code and produce a graph from it was started using PL/I. The procedure looks promising and will be further studied in the future.

7.5 Correlation of Program Errors with Complexity Measures

In order to assess the validity of complexity measures advanced in Refs. 9 and 27, a data collection has been planned using an IBM 370-125 computer operating under DOS. All output generated by the computer is stored on the output queue of the operating system generated file POWERQ. The process is cumulative and continues until POWERQ is filled, which takes approximately one week.

As a next step the POWERQ output file will be run through a program which scans it for FORTRAN programs. When a FORTRAN program is found, the program copies the job card information, then continues scanning for syntax and run time errors. These are identified by the prefix ILF in the output queue. There are a total of 301 such errors detectable by the system. The errors are copied after the job card information. The process continues until the entire POWERQ file has been scanned. The sought-after information is copied onto a tape.

The resulting tape will then be sorted according to job number, name and program number, which will provide unique identification of the program. After sorting, the tape will be merged with a master tape on which the following information will be accumulated:

1. A count of all runs which contain syntax errors. Syntax errors are those with error numbers from 1 to 206.

2. A count of all runs which contain run time errors. These are identified by error numbers 207 to 301.

3. A count of all error free runs. It will be assumed that the number of logical errors is one less than the number of error free runs.

4. A total count of all runs.

The final version of each program will be collected from the students and read onto a tape using the job card with which it was run. These programs will then be run through the modified version of a program obtained from Professor Halstead, which counts frequency and other parameters.

Using the frequency counts (F_i), the entropy measure (i.e., $\sum f_i \log_2 p_i$) will be obtained and stored, together with job card information, on a tape. The probabilities p_i will be obtained from:

1. previously run programs

2. the current programs which are being tested.

The tape so generated will eventually be sorted. This tape, containing the complexity measure, will then be compared to the error counts, and a correlation determined, if one exists.

Most of the programming effort to date has been spent on the program needed to handle the POWERQ file. Methods were studied to access and process this file. The program is now almost ready and only minor changes remain. This work will be a part of a forthcoming Ph.D. thesis.

7.6 Choice of Strategy in Software Revision

For a wide variety of reasons, software undergoes changes and revisions during its life cycle. Examples of such reasons are:

1. A new or changed specification has to be incorporated.

2. The host computer or source language for the software has been changed.

3. Peripheral equipment has been changed.

4. Significant bugs are found in the deployed software.

5. Format changes in input or output are required.

6. The program is to be used as a module in different software and the interfaces must be matched.

When confronted with such changes, the first decision concerns the choice among the following approaches:

1. should the code be discarded and new code written,

2. should the code be retained, but major modification be made,

3. should only minor modifications be made.

The choice among these three alternatives is predicated upon: the quality of the software under consideration, the extent and significance of the required change, the expected life-time of the new software, and the budgeted resources. The selection of alternative has to be made on a technical and managerial basis. At present there are few objective tools to aid in such decisions.

Early work in this area by S. Amster (Ref. 36) correlated various subjective and objective measures of program quality. The subjective measures elicited opinions on the extent of changes needed to improve clarity, decrease memory size, and decrease run-time. The objective measures included such factors as number of lines of code, number of GoTo's, number of calls, and so on.

More recently M. Halstead (Ref. 27, Chapter 7) has classified several so-called program "impurities," which detract from clarity. The elimination of these impurities improves the program readability.

We are investigating the use of Amster's and Halstead's techniques, as well as others, to produce a ranking to aid the designer and the manager in their decision.

7.7 Effect of Organizational Structure on Software Development

On all large technological tasks, the organizational structure of the project team has a large influence on the productivity, reliability, and quality of the product. These effects are especially strongly exhibited in software development (see Reference 39). A major phenomenon of interest is that productivity is often not proportional to man hours. This fact has been observed by many and articulated by Brooks (see Reference 37). In fact, at some stages of software development, adding workers to the project slows the pace of progress.

We have initiated research in this area by studying the research literature on graph theoretic models of organizational behavior. Much of the qualitative literature relates to research done by psychologists prior to 1950. This work, which supplies graph structures for various group organizational structures, is summarized in Reference 38.

A simplistic assumption is usually made that programming productivity is a direct function of charged time. However, this is not a satisfactory assumption because charged time represents raw man hours composed of personal time (coffee breaks, conversations, etc.),studying time, communication time, and lastly, productive time. We can assume that the proportion of personal time is fixed at say 10% regardless of the organizational structure. However, the remaining time divisions are highly dependent on the organizational structure. Preliminary results indicate (see Ref. 40) that we are able to model the communication links as paths in the graph structure, and the constraints (a fixed number of man hours or a fixed number of programmers). The object is to evaluate the merits of various organization schemes so as to minimize the need for communication and thus increase the productivity.

A simple example illustrates the phenomenon of increased man hours resulting in decreased productivity. Suppose 10 men work on a project, and their total of 400 hours per week are spent on 40 hours of personal time, 160 hours of studying and communications time, and 200 hours of productive time. A programmer is added to the group, and since he is new, he spends his 40 hours on 4 hours of personal time, 16 hours of productive time, and 20 hours on studying and communications. Thus, he adds 16 hours of productive time to the task. However, suppose that the integration of this new man into the group requires 1 hour review of the project attended by all. This is a group loss of 10 hours of productive time. In addition, the new man is assigned to spend a day (8 hours) with an experienced team member to help him get started on his assigned task. Thus, another 8 hours are lost, and the net result of this added worker is a two hour loss.

Work is presently continuing on establishing further relationships among the organizational structure and productivity, reliability and quality, and of designing a realistic model of this task.

7.8 Collection, Storage, and Retrieval of Software Reliability Data

The collection, storage, and availability of reliability data is of vital importance to the worker in the field of software reliability. The data is needed by experimentalists to suggest models. It is needed by theoreticians to test the hypothesis of their postulated models and to evaluate their parameters. Finally, it is needed by managers and designers as a data base for design decisions.

A preliminary study supported by RADC (Ref. 31) discusses the scope, specific techniques, and feasibility of establishing a software reliability data base. In the course of writing material on software engineering (Ref. 21), the following rough estimates were made on the amount of information generated annually in writing software for the Air Force.

a. The total number of bugs/year is roughly between $7.5 * 10^5$ and $7.5 * 10^6$.

b. Assuming that ten data descriptors/bug must be stored, one needs between $7.5 * 10^6$ and $7.5 * 10^7$ words of storage. All these words may be stored on a few reels of tape and disks.

d. Assuming microfiche is used for permanent backup storage, one needs between $7.5 * 10^3$ and $7.5 * 10^4$ microfiche.

Further details and the underlying assumptions are given in Reference 21.

7.9 Classification and Enumeration of Program Errors

In order to obtain some preliminary quantitative data on the various types of programming errors, approximately 1,000 programs were analyzed manually. In this work, described in Ref. 1, we classified the errors into

three broad categories, namely, syntax, semantic, and algorithmic. Within these three categories, we enumerated 117 different errors. Rather than listing individual numbers of occurrences, we rated the errors on the scale from 1 (typifying a rare occurrence) to 5 (for a frequent occurrence).

The desired statistics were obtained from programs in a first and in a second programming course for the following three reasons:

1. The student programs were under our control.

2. In a typical programming assignment given to a class of 25 students, the same program is run approximately 50-100 times (about 2-4 runs per student, with the extra runs for debugging or cosmetic improvements). Since the assignments are known, we know the results and can note the presence of algorithmic errors, a task that will be impossible in a "strange" program.

3. The student programs are a good sample because the number of errors is nearly constant in a programming assignment. This is so because at the start of the semester, even though the students are inexperienced, the assignments are simple and have a few lines of code. Later in the semester the students are more experienced and make fewer errors per line of code, but the assignments are harder and require more lines of code.

We tried to gather various statistics on programming errors by issuing forms to students and asking them to record various program bugs. However, this method was not successful. Some students were suspicious that these forms were used for grading purposes. Other students did not want to admit to a high number of errors or runs. Consequently, we abandoned this voluntary method. What we did instead was to modify the output for the student programs so as to produce two identical printouts of each program. One was returned to the student, and the other was retained for us in the computer center. We collected these duplicate programs and used them in our analysis. This duplication was made possible by the fact that the students' programs, written in the PL/I language, are compiled with the PLAGO interpreter (which implements a subset of PL/I) developed here and completely under our control

To compile the statistics we used the form described in Reference 1. Such forms were compiled for each student. From these forms we enumerated the number of different errors. For details we again refer to Reference 1.

The tediousness of the manual analysis and the large volume of programs to be recorded lead to plans for automatic collection. Two such plans are described in the next sections (Sections 7.10 and 7.11).

7.10 Modifications of the PLAGO Interpreter

The manual classification of compile and execution time errors was discussed in Section 7.9. This work indicated the need for automatic collection for the following two main reasons:

1. The volume of errors was too large for manual collection and reduction.

2. The students' reporting data was not always reliable. These forms often reported fewer errors than actually occurred, because they thought that such good results will please the instructor.

Consequently an automatic collection scheme was undertaken. The purpose of the collection was to obtain the following information:

1. The number of PLAGO programs submitted. PLAGO is a PL/I dialect containing a scientific subset of PL/I, developed at the Polytechnic and used in our programming courses. PLAGO was chosen as the subject language, because it is entirely in our control.

2. The number of errors in each category. Such recording is possible, because each category is identified by an error number. These categories include both compile and execution time errors.

3. Certain logical errors occurring during execution. This can be done by requiring students to use prescribed variable names in programming assignments and recording the results which are different from the correct ones.

All these tasks were undertaken as a Master Thesis by a student. Unfortunately, he took a leave of absence which continues at this time.

A similar effort, using the FORTRAN compiler, is described in Section 7.5. Nevertheless, we hope to continue the PLAGO error collection in the future either with the returning student or with other personnel.

7.11 Small Scale Tests

In order to study, verify, and measure parameters of the theoretical models developed (Refs. 8, 9, and 14), small scale tests were planned and began. The tests consisted of four tasks, programmed by student programmers, with careful records kept on their debugging experience. The four tasks and the reporting form are described in Reference 5 (pp. 22-31).

The student programmers were of sophomore-junior standing, with high interest and ability in programming topics. Because of this selectivity we believe their product to be equivalent to programs produced by programmers with intermediate experience. Consequently, we consider the obtained test data to be representative of normal practice.

The programmers were made aware of the importance of maintaining careful and truthful records. They were also given the following specific instructions:

1. The problems were to be analyzed and coded, with both analysis and coding times recorded.

2. The initial program was to be corrected of just the syntax errors. Their number, number of runs needed for their correction, and run times were to be recorded. All printouts were to be saved and numbered.

3. The programs were presented to us (i.e., to either M. Shooman or H. Ruston) for inspection. We then asked the program author and other members of the group to debug each copy independently, recording:

   a. Number of bugs and types found in each debugging shot.

   b. Analysis time and computer time for each debugging shot.

   c. History of removed bugs and generated bugs.

4. A programmer who reached a blind alley had to consult with us. He could neither ask for other help, nor abort the program.

5. The programs were constructed with the following constraints:

   a. Structured design was to be followed.

   b. The instructions were to contain no impurities (Reference 27).

   c. The main program was to be the control structure, with calls to modules (i.e., blocks or procedures) for various tasks.

   d. No module was to exceed 50 lines.

The first three programs were written by five different student programmers. Most have been completely debugged and documented. We must still debug the remaining programs, record data on the fourth program, and analyze the data. Present plans envision the continuation of this effort during the summer of 1978, when student programmers are available.

7.12 Automatic Programming Techniques

This study addresses itself to three questions, namely,

a. Can a better applications program design be obtained if certain program blocks are available in a computer library? (By better we mean all the goodness factors of a design - less cost, more reliability, etc., however, we do not necessarily count run time or memory size as long as they are within reason).

b. If the answer to the first question is in the affirmative, what blocks should be written and placed in the library,

and

c. how is the information about these blocks to be transmitted to the program designer?

In the research to be described, we assume that the answers to these three questions are:

a. If program blocks are already available, the incorporation of these blocks certainly simplifies the design task. This statement is evidenced by the fact that to design around available code is the standard design technique. A further advantage of reusing available code is that inexperienced junior programmers may produce reliable software, if critical parts have been designed by better or more experienced programmers. Also, many applications programs are written by personnel more experienced with the application than with programming techniques, and such a system might significantly improve the quality of their programs.

b. The question regarding block contents can only be answered in a general way. If we accept the hypothesis that there exists a high degree of commonality in classes of commercial and scientific applications, then a set of blocks can be defined whose members appear frequently in most designs for such classes of applications. Clearly, different blocks are needed for those writing payroll software than for the designers of a wind tunnel simulation. Hence blocks must be grouped in certain sets which form a library. Each library applies to related applications.

c. The mode of transmittal of the library is a crucial one. A listing of available programs in some volume is of little use to an inexperienced programmer, because such programmer may not know, for example, that he can utilize program #123 to advantage in his design. Just the title of such a program (say Gram-Charlier interpolation) may impact no meaning to our designer. Even if this program has a short decription of its use, often such description is for the practitioner rather than for the tyro.

In view of these questions and the stated answers, a technique has been investigated, called here automatic programming. Our implementation of this technique is an interactive terminal system which allows on-line queries.

This work, described in Refs. 3, 4, and 5, consists of two major units, the programs FLOW and AUTO. The program FLOW uses four types of blocks to generate a flowchart of the program. The blocks are:

1. The Control Block. This is a conditional decision block and is written by the programmer.

2. The Functional Block. This is the library block to be used in the design.

3. The Stop Block. This block signals the end of the path and is coded by the STOP statement.

4. The User's Code Block. This block contains the code written by the programmer, exclusively for the application at hand.

Upon completion of the flowchart generation, the control passes to the second unit, that is, to the program AUTO.

The purpose of the program AUTO is to aid the program designer in the choice of the library member. If there are several such suitable library programs, AUTO will help in the selection of the most appropriate one.

How is this done? The user will specify to AUTO what he likes to do, and AUTO will advise which of the available programs do this. AUTO will also tell the characteristics of each method (e.g., will tell that trapezoidal formula approximates each pair of points of the function by straight line segments) in response to each query.

This work forms the nucleus of a Ph.D. dissertation presently in progress.

8.0 References

1. M. Shooman and H. Ruston, "Summary of Technical Progress - Software Modeling Studies," Contract No. F-30602-74-C-0294, Polytechnic Institute of New York, EE/EP74-019, EER 114, April 1974-September 1974.

2. M. Shooman and H. Ruston, "Summary of Technical Progress - Software Modeling Studies,", Polytechnic Institute of New York, Interim Report (Oct 74 - Jun 75), RADC-TR-75-245, Sep 1975, AD# A018 618.

3. M. Shooman and H. Ruston, "Summary of Technical Progress - Software Modeling Studies," Polytechnic Institute of New York, Interim Report (Jul 75 - Dec 75), RADC-TR-76-143, May 1976. AD# A025 895.

4. M. Shooman and H. Ruston, "Summary of Technical Progress - Software Modeling Studies,", Polytechnic Institute of New York, Interim Report (Jan 76 - Jun 76), RADC-TR-76-405, Jan 1977. AD# A036 721.

5. M. Shooman and H. Ruston, "Summary of Technical Progress - Software Modeling Studies," Polytechnic Institute of New York, Interim Report RADC-TR-75-169, Jul 1975, AD# A038 508.

6. A.K. Trivedi and M.L. Shooman, "Computer Software Reliability: Many-State Markov Modeling Techniques," Polytechnic Institute of New York, Interim Report, RADC-TR-75-169, Jul 1975, AD# A014 824

7 M. Shooman and S. Natarajan, "Effect of Manpower Deployment and Bug Generation on Software Error Models," Polytechnic Institute of New York, Interim Report, RADC-TR-76-400, Jan 1977. AD# A036 106.

8. B. Rudner, "Seeding/Tagging Estimates of the Number of Software Errors: Models and Estimates," Polytechnic Institute of New York, Interim Report, RADC-TR-77-15, Jan 1977, AD# A036 655.

9. A. Laemmel and M.L. Shooman, "Statistical (Natural) Language Theory and Computer Program Complexity," POLY EE/EP 76-020, SMART 107, August 1977.

10. A. Laemmel, "Study of Recursive Function Theory and Its Application to Program Complexity," POLY EE77-037 SRS 108, September 1977.

11. G.S. Popkin, "On the Number of Tests Necessary to Verify a Computer Program," POLY EE77-039, SRS 109, October 1977.

12. D. Baggi and M.L. Shooman, "An Automatic Driver for Pseudo-Exhaustive Software Testing," submitted for 1978 IEEE Spring Compcon Conference, San Francisco, California 1978.

13. M.L. Shooman, "Software Reliability: Measurement and Models," 1975 Annual Reliability and Maintainability Symposium.

14. M.L. Shooman, "Operational Testing and Software Reliability Estimation During Program Development," 1973 IEEE Symposium on Computer Software Reliability, April 30, 1973, New York

15. M.L.Shooman, "Probabilistic Models for Software Reliability Prediction," Statistical Methods for the Evaluation of Computer System Performance, Frieberger, Editor, Academic Press, New York, 1972.

16. M.L. Shooman, "Structural Models for Software Reliability Prediction," Second National Conference on Software Engineering, October 1976, San Francisco.

17. H.D. Mills, Internal Memorandum, IBM Federal Systems Division, September 1970.

18. Morton Hyman, personal communication, IBM Federal Systems Division, Morris Plains, New Jersey.

19. M.L. Shooman and S. Sinkar, "Generation of Reliability and Safety Data by Analysis of Expert Opinion," Procededings, 1977Annual Reliability and Maintainability Symposium, Philadelphia, Pa. (Received best paper award.)

20. S.N. Mohanty, "Automatic Program Testing," Ph.D. Dissertation, Polytechnic Institute of New York, 1976.

21. M.L. Shooman, "Software Engineering: Reliability, Design, Management," CS606 Class Notes, Polytechnic Electrical Engineering Department, Fall 1977.

22. Denis Baggi and M.L. Shooman, "Test Models: Classification and Automatic Driver Design," POLY EE77-040, SRS 110, November 1977.

23. A.Laemmel, "Statistical Test Models," POLY EE77-041, SRS 111, December, 1977.

24. M. Shooman, "Meaning of Exhaustive Software Testing," PINY EE/EP 74-006, EER 105, January 1974.

25. M. Shooman and A. Laemmel, "Statistical Theory of Computer Programs - Information Content and Complexity," Digest of Technical Papers, IEEE Compcon, Fall 1977 (Received best paper award) pp. 341-347.

26. S.N. Mohanty and M. Adamowicz, "Proposed Measures for the Evaluation of Software," Proceedings of the MRI Symposium on Computer Software Engineering, April 1976.

27. M.H. Halstead, "Software Science," Elsevier, North-Holland, New York, New York, 1977.

28. A.N. Sukert, "A Software Reliability Modeling Study," RADC-TR-76-24, In-house report, August 1977, AD A024 068.

29. A.N. Sukert, "An Investigation of Software Reliability Models," Proceedings 1977 Annual Reliability and Maintainability Symposium, IEEE, New York, N.Y. p. 478ff.

30. M.L. Shooman, J. Johnsen, and R. Straub, "Generation of Failure Data Fitting a Specified Hazard Function," Ninth Reliability and Maintainability Conference, Detroit, Mich., July 1970, Proceedings of Society of Automotive Engineers, Inc., New York, N.Y., pp. 405-413.

31. L. Duval, "Software Data Repository Study," Report RADC-TR-76-387, Rome Air Development Center Griffiss Air Force Base, New York, 13441, December 1976.

32. T.J. McCabe, "A Complexity Measure," IEEE Transactions of Software Engineering, December 1976, p. 308.

33. C. Marshall, "Applied Graph Theory," John Wiley, New York, 1971.

34. D. Minoli, "Combinatorial Graph Complexity," Accademia Nazionale Dei Lincei, Estratto dai Rendiconti della Classe di Scienze fisiche, mathematiche e natruali, Serie VIII, Vol. LIX, December 1975.

35. R. Flynn, "Dynamic Complexity of Networks," Ph.D. Dissertation, Department of Mathematics, Polytechnic Institute of Brooklyn, 1973.

36. S. Amster, et al, "An Experiment in Automatic Quality Evaluation of Software," Proceedings of the 1976 Polytechnic Symposium on Computer Software Engineering, John Wiley and Sons, New York, 1976.

37. F.P. Brooks, Jr., "The Mythical Man-Month," Addison-Wesley Pub. Co., Reading, Mass., 1975, pp. 83-90.

38. H.H. Goode and R.E. Machol, "System Engineering," McGraw-Hill, New York, 1957, Section 25-2, Group Dynamics, pp. 383-393.

39. G. Weinberg, "The Psychology of Computer Programming," Reinhold-Van Nostrand, New York, 1971.

40. M.L. Shooman, "Development of a Structural/Psychological Model of Programmer Productivity and Correctness," Internal memorandum, March 17, 1977.

9.0 Professional Activities

The results of the research described in this report, have been disseminated both in preliminary and in completed forms. The professional activities are grouped below under the following caegories: (1) Papers, (2) Reports, (3) Symposia, (4) Talks and Seminars, (5) Books, (6) Committees, (7) Theses, and (8) Professional Awards.

9.1 Papers

The following lists the papers published in Journals and Conference Proceedings during the period of this contract.

1. M.L. Shooman and S. Tenenbaum, "Hazard Function Monitoring of Airline Components," 1974 Annual Reliability and Maintainability Symposium, January 1974.

2. R. Schatz, M. Shooman and L. Shaw, "Time Dependent Stress-Strength Models of Non-Electrical and Electrical Systems," 1973 Annual Reliability and Maintainability Symposium, January 1973.

3. E. Cohen and H. Ruston, "A Class of Optimal and Polynomials Related to Electric Band-Pass Filters," SIAM Journal on Applied Mathematics, Vol. 26, No. 2, March 1974, pp. 396-412.

4. M.L. Shooman, "Software Reliability: Measurements and Models," Proceedings 1975 Annual Reliability and Maintainability Symposium, January 28-30, 1975.

5. R. Flynn, "Design of Computer Software," Proceedings 1975 Annual Reliability and Maintainability Symposium, January 28-30, 1975.

6. A.K. Trivedi and M.L. Shooman, "A Many-State Markov Model for the Estimation and Prediction of Computer Software Performance Parameters." Proceedings 1975 International Conference on Reliable Software, April 1975.

7. M.L. Shooman and M.I. Bolsky (Bell Laboratories), "Types, Distribution, and Test and Correction Times for Programming Errors," Proceedings 1975 International Conference on Reliable Software, April 1975.

8. S.N. Mohanty and M. Adamowicz, "Proposed Measures for Evaluation of Software," Proceedings MRI Symposium on Computer Engineering, New York City, April 1976.

9. M.L. Shooman and S. Natarajan, "Effect of Manpower Deployment and Error Generation on Software Reliability," Proceedings MRI Symposium on Computer Software Engineering, New York City, April 1976.

10. M. Shooman, "Software Reliability: Analysis and Prediction," Transactions 14th Annual Long Island Section ASQC Conference, April 10, 1976.

11. L. Shaw and S. Sinkar, "Redundant Spares Allocation to Reduce Reliability Costs," Naval Research Logistics Quarterly Vol. 23, No. 2, pp. 179-194, June 1976.

12. M.L. Shooman and M.I. Bolsky, "Types, Distribution, and Test Correction Times for Programming Errors," IEEE Transactions on Reliability, Vol. R-25, No. 2, pp. 69-70, June 1976.

13. M.L. Shooman and A.K. Trivedi, "A Many-State Markov Model for Computer Software Performance Parameters," IEEE Transactions on Reliability, Vol. R-25, No. 2, pp. 66-68, June 1976.

14. M.L. Shooman, M. Horodniceanu, and E.J. Cantilli, "System Safety Applied to Transportation Systems," Proceedings 1976 Inter Society Conference on Transportation, Los Angeles, Calif., July 1976.

15. M.L. Shooman, "Recent Developments in Software Reliability - The State of the Art," Proceedings of the Thirteenth IEEE Computer Society International Conference, Washington, D.C., September 1976.

16. M.L. Shooman, "Structural Models for Software Reliability Prediction," Second National Conference on Software Engineering, October 1976, San Francisco, Calif.

17. M.L. Shooman, "Software Error Models - Summary of the State of the Art," Proceedings Second National Conference on Software Engineering, Berkeley, Calif., October 1976.

18. L. Shaw and M. Shooman, "Confidence Bounds and Propagation of Uncertainties in Systems Availability and Reliability Computations," Naval Res. Logistics Quarterly, Vol. 24, No. 4, pp. 673-685, December 1976.

19. M.L. Shooman and S. Sinkar, "Generation of Reliability and Safety Data by Analysis of Expert Opinion," Proceedings 1977 Annual Reliability and Maintainability Symposium, Philadephia, Pa., January 1977.

20. M. Shooman, "Software Reliability Models and Measurement," Infotech State of the Art Conference on Reliable Software, Proceedings, London, England, March 1, 1977.

9.2 Reports

The following lists the technical and summary reports published during the duration of this contract..

1. M. Shooman, "Meaning of Exhaustive Software Testing," PINY EE/EP 74-006, EER 105, January 1974.

2. A.K. Trivedi and M.L. Shooman, "A Markov Model for the Evaluation of Computer Software Performance," Polytechnic Institute of New York Research Report No. EE/EP 74-011-EER 110, supported by RADC Contract No. F-30602-74-C-0294 and ONR Contract No. N00014-67-A-0438-0013, May 1, 1974.

3. L. Shaw and J. Hsuan, "Control and Dynamic Repairman Assignment for Stochastic Linear Systems," Polytechnic Institute of New York, Poly EE/EP-74-013, May 1974.

4. M. Shooman and H. Ruston, "Summary of Technical Progress, Software Modeling Studies," Contract No. F-30602-74-C-0294, Polytechnic Institute of New York, EE/EP-74-019, EER 114, April 1974-September 1974.

5. A.K. Trivedi and M.L. Shooman, "Computer Software Reliability: Many-State Markov Modeling Techniques," Polytechnic Institue of New York, Interim Technical Report, RADC-TR-75-169, July 1975, AD A014 824.

6. L. Shaw, Final Report on NSF Grant GK40185, "Optimal Control of Queues and Other Markov Systems," February 1975.

7. M. Shooman and H.Ruston, "Summary of Technical Progress - Software Modeling Studies," October 1974-June 1975, Polytechnic Institute of New York, Interim Technical Report, RADC-TR-77-245, July 1977.

8. M. Shooman and H. Ruston, "Summary of Technical Progress, Software Modeling Studies," July 1975-December 1975, Polytechnic Institute of New York, Interim Technical Report, RADC-TR-76-143, March 1976, AD A025 895.

9. L. Shaw and M. Shooman, "Confidence Bounds and Propagation of Uncertainties in System Availability and Reliability Computations," Technical Report N00014-67-A-0438-0013, Poly EE/EP-75-002 Polytechnic Institute of New York, January 1976.

10. C.L. Hsu and L. Shaw, "Downtime Distributions Based on a Multivariate Exponential Distribution," Report No. Poly EE/EP-76-002 EER 120, Polytechnic Institute of New York, February 1976.

11. C. Marshall, "Contributions to the Theory of Availability," Report No. Poly EE/EP-76-004 EER 121, Polytechnic Institute of New York, February 1976.

12. M. Horodniceanu, E. Cantilly, M. Shooman, and L. Pignataro, "Transportation System Safety - A Literature Survey and Annotated Bibliography," Report March 1976, U.S. Dept. of Transportation Contract DOT-05-50241.

13. M. Shooman, and S. Natarajan, "Effect of Manpower Deployment and Bug Generation on Software Error Models," Polytechnic Institute of New York, Interim Technical Report, RADC-TR-76-400, January, 1977, AD A036 106.

14. M. Shooman, and H. Ruston, "Summary of Technical Progress, Software Modeling Studies," January 1976-June 1976, Polytechnic Institute of New York, Interim Technical Report, RADC-TR-76-405 January 1977, AD A038 299.

15. B. Rudner, "Seeding/Tagging Estimates of the Number of Software Errors: Models and Estimates," Polytechnic Institute of New York, Interim Technical Report, RADC TR-77-15, January 1977, AD A036 655.

16. M. Shooman, and H. Ruston, "Summary of Technical Progress - Software Modeling Studies," July 1976-December 1976, Polytechnic Institute of New York, Interim Technical Report, RADC-TR-77-88, March, 1977, AD A038 508.

17. M. Shooman, and S. Sinkar, "Generation of Reliability and Safety Data by Analysis of Expert Opinion," Poly EE/EP Report, January 1977.

18. A. Laemmel and M.L. Shooman, "Statistical (Natural) Language Theory and Computer Program Complexity," POLY EE/EP-76-020, SMART 107, August 1977.

19. A. Laemmel, "Study of Recursive Function Theory and Its Application to Program Complexity," POLY EE77-037 SRS 108, September 1977.

20. G.S. Popkin, "On the Number of Tests Necessary to Verify a Computer Program," POLY EE77-039, SRS 109, October 1977.

21. Denis Baggi, and M.L. Shooman, "Test Models: Classification and Automatic Driver Design," POLY EE77-040, SRS 110, November 1977.

22. A. Laemmel, "Statistical Test Models," POLY EE77-041, SRS 111, December 1977.

9.3 Symposia

The following lists the symposia participation by the project personnel.

1. M.L. Shooman, Organizer of a Session on Software Reliability for the 1975 Annual Reliability and Maintainability Symposium, January 30, 1975.

2. M.L. Shooman, General Co-Chairman, 1975 International Conference on Reliable Software, Los Angeles, California, April 1975.

3. M.L. Shooman, Chairman, Program Committee, MRI Symposium on Computer Software Engineering, New York City, April 1976.

4. M. Adamowicz, A. Laemmel, and H. Ruston, Members, Program Committee, MRI Symposium on Computer Software Engineering, New York City, April 1976.

9.4 Talks and Seminars

The following lists the participation by the personnel in talks and seminars.

1. Talk by Dr. Hans Tarboux of NASA Marshall Space Flight Center, to and with the participation of the group personnel, January 1974.

2. M.L. Shooman, Lectured in UCLA special course on software engineering, January 21, 22, 1974.

3. Talk by M.L. Shooman on Models for Software Reliability, Measuring and Testing, at Joint Serices Electronics Program Topical Review, University of Texas, March 1974.

4. M.L. Shooman, Joint Services Electronics Program Meeting, Univ. of Texas at Austin, March 19, 1974.

5. Participation by M.L. Shooman at the Workshop on the Attainment of Reliable Software, University of Toronto, June 1974.

6. Talk by M.L. Shooman on the Capability of Software to Enhance (or Degrade) Total System Reliability, American Management Association Conference Program, September 1974.

7. M.L. Shooman, Delivered talk at conference on Fault Tree Analysis at U.C. Berkeley, September 3, 4, 1974.

8. Chairing of a Session by M.L. Shooman on Computer Verification and Reliability, American Management Association Conference Program, September 1974.

9. Talk by M.L. Shooman on Software Reliability Modeling at Syracuse University in Fall of 1974.

10. A. Trivedi, Talk on Reliability Analysis at A.I.Ch.E. Annual Meeting, Washington, December 2, 1974.

11. M.L. Shooman, Joint Services Electronics Program Meeting, talk on Software Availability Models, December 3, 1974.

12. M.L. Shooman, "Analytical Safety and Reliability," Joint meeting of Ottawa chapter of the IEEE and the Society of Reliability Engineers, January 8, 1975.

13. M.L. Shooman, "Software Reliability: Measurements and Models," 1975 Annual Reliability and Maintainability Symposium, January 28-30, 1975.

14. R. Flynn, "Design of Computer Software," 1975 Annual Reliability and Maintainability Symposium, January 28-30, 1975.

15. H. Ruston, "Structured Programming," Computer Seminar, PINY, February 1975.

16. S. Habib, "Specification Languages," Computer Seminar, PINY, November 1975.

17. M.L. Shooman, Joint Services Electronics Program Meeting, Talk on Software Availability Models, December 1975.

18. M.L. Shooman, H. Ruston, M. Adamowicz, A. Laemmel, and B. Rudner, "Software Engineering Topics," Seminar, PINY, December 1975.

19. M.L. Shooman, and H. Ruston, Presentation at RADC, December 1975.

20. H. Ruston, "Structured Programming," Computer Seminar, PINY, January 1976.

21. S. Habib, "An Overview of Microprocessors," Seminar, PINY, February 1976.

22. H. Ruston, "Top-down Design," Computer Seminar, PINY, March 1976.

23. D. Baggi, "Design of Automatic Test Drivers," Seminar, PINY, June 1976.

24. S. Habib, "User Services in Remote Entry Environment," National Science Foundation Conference on Computers in Undergraduate Education, Binghamton, N.Y., June 1976.

25. H. Ruston, "Structured Programming with PL/1 and FORTRAN Applications," Seminar, PINY, August 1976.

26. S. Habib, "Computer Hardware Organization for Programmers," Seminar, PINY, September 1976.

27. M.L. Shooman, "Recent Developments in Software Reliability - The State of the Art," Thirteenth IEEE Computer Society International Conference, Washington, D.C., September 1976.

28. M.L. Shooman, "Structural Models for Software Reliability Prediction," Second National Conference on Software Engineering, October 1976, San Francisco, California.

29. M.L. Shooman, and H. Ruston, "Cost Reducing, High Reliability Programming Techniques," 1976 ORSA/TIMS Joint National Meeting, November 1976.

30. M.L. Shooman, and S. Sinkar, "Generation of Reliability and Safety Data by Analysis of Expert Opinion," 1977 Annual Reliability and Maintainability Symposium, Philadelphia, Pennsylvania.

31. M. Shooman, "Program Complexity, Run-Time, and Storage Models," invited for presentation at Annual Spring ORSA/Tims Conference, San Francisco, May 9-11, 1977.

9.5 Books and Notes

1. S. Amster, and M.L. Shooman, "Software Reliability: An Overview," published in Reliability and Fault Tree Analysis, edited by R. Barlow et al., SIAM, Philadelphia, 1975.

2. M.L. Shooman, "Software Reliability: Analysis and Prediction," published in Generic Techniques in Systems Reliability Assessment, edited by E. Henley and J. Lynn, Noordhoff International, Reading, Massachusetts, 1976.

3. M.L. Shooman, "Software Engineering: Reliability, Design, Management," Notes for CS606, Polytechnic Institute of New York, Dept. of Elec. Engineering, Fall 1977.

4. H. Ruston, "Programming with PL/I," McGraw Hill Book Co., New York, 1978.

9.6 Technical Committees

1. M.L. Shooman, Member IEEE ADCOM (Governing committee) of the Group on Reliability.

2. M.L. Shooman, Co-Chairman, IEEE Computer Society Technical Committee on Software Engineering.

3. M.L. Shooman, Consumer Member, Consumer Product Safety Commission's Working Group on Safety Standards for Power Lawn Mowers.

4. M.L. Shooman, Member Software Reliability Group, Joint Logistic Commanders Reliability Workshop.

5. M.L. Shooman, Member, National Academy of Sciences Technological Trade-offs Advisory Committee to the Air Force Systems Command.

6. M.L. Shooman, Member of NASA Advisory Committee on Guidance, Control and Information Systems.

7. M.L. Shooman, Chairman, Program Committee, Session Chairman, MRI Symposium on Computer Software Engineering, New York, NY, April 1976.

8 M.L. Shooman, Member, Executive Committee, IEEE Computer Society Technical Committee on Software Engineering.

9. M. Adamowicz, S. Habib, A. Laemmel, and H. Ruston, Members, Program Committee, MRI Symposium on Computer Software Engineering, New York, N.Y., April 1976.

10. S. Habib, Chairman, National Lectureship Committee of the Association for Computing Machinery.

11. S. Habib, Chairman, SIGMICRO (Special Interest Group on Microprogramming) of ACM.

9.7 Theses

The following four dissertations were published by the project's personnel:

1. A.K. Trivedi, "Computer Software Reliability: Many-State Markov Modeling Techniques," June 1975.

2. J. Hsuan, "Optical Control and Dynamic Repairman Assignment for Stochastic Linear Systems," Dept. of Operation Research, Polytechnic Institute of New York, June 1975.

3. L.R. Doyon, "Markov Chains in Reliability Analysis by Computer," Dept. of Operation Research, Polytechnic Institute of New York, June 1975.

4. S.N. Mohanty, "Automatic Program Testing," Dept. of Electrical Engineering, Polytechnic Institute of New York, June 1976.

The following Master of Science thesis was published by the project personnel:

S. Natarajan, "Effect of Manpower Deployment and Bug Generation on Software Error Models," Dept. of Electrical Engineering, Polytechnic Institute of New York, June 1976.

9.8 Professional Awards

1. Professor M.L. Shooman has been chosen (with S. Sinkar) as winner of the 1977 P.K.McElroy Award for best technical paper at the Annual Reliability Symposium (see Ref. 12). Dr. Shooman has won this award previously in 1967 and 1971, making him the only three-time winner in the 22 years of the Symposium.

2. Professors M.L. Shooman and A. Laemmel were winners of the Best Technical Paper Award at the IEEE Computer Societies COMPCON Conference in September 1977.

10.0 Personnel

During the course of this three-year study, the following Polytechnic faculty, staff, and students contributed to the research effort of this contract:

	1974	1975	1976	1977
Principal Investigator				
Martin L. Shooman	X	X	X	X
Faculty Investigators				
Henry Ruston	X	X	X	X
Michael Adamowicz	X	X	X	-
Denis Baggi	X	X	X	X
Stanley Habib	-	X	X	-
Arthur Laemmel	X	X	X	X
Clifford Marshall	-	-	X	-
Staff				
David Doucette	X	X	.	-
Beulah Rudner	X	X	X	-
Sandresh Sinkar	-	X	-	-
Students				
Kenneth Apperley	-	-	X	-
Marek Babinski	X	X	-	-
Eli Berlinger	-	-	X	X
John Casey	-	-	X	-
Daniel Kaufman	-	-	X	-
Ronald Karam	-	-	X	-
Arthur Law	X	X	-	-
Robert Leone	-	X	-	-
Harry Linzer	-	-	X	-
Ellan Lipschitz	X	X	X	-
Siba Mohanti	X	X	X	-
Srinivasan Natarazan	-	X	X	-
Garry Popkin	X	X	X	X
Susan Shmoys	-	-	X	-
Ashok Trivedi	X	X	-	-