AD-A038 508 POLYTECHNIC INST OF NEW YORK BROOKLYN DEPT OF ELECTR--ETC F/G 9/2
SUMMARY OF TECHNICAL PROGRESS, SOFTWARE MODELING STUDIES.(U)
MAR 77 M L SHOOMAN, H RUSTON F30602-74-C-0294

UNCLASSIFIED POLY-EE/EP-76-023 RADC-TR-77-88 NL

| OF |
AD
A038508

END

DATE
FILMED
5-77

ADA038508

RADC-TR-77-88
Technical Report
March 1977

SUMMARY OF TECHNICAL PROGRESS, SOFTWARE MODELING STUDIES

Polytechnic Institute of New York

Approved for public release; distribution unlimited.

AD No.
DDC FILE COPY

DDC
RECEIVED
APR 20 1977
D

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.

This report has been reviewed and approved for publication.

APPROVED: Alan N. Sukert

ALAN N. SUKERT, Captain, USAF
Project Engineer

APPROVED: Robert D. Krutz

ROBERT D. KRUTZ, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

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-77-88	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: SUMMARY OF TECHNICAL PROGRESS, SOFTWARE MODELING STUDIES	5. TYPE OF REPORT & PERIOD COVERED: Interim Report, 1 Jul 76 - 31 Dec 76
	6. PERFORMING ORG. REPORT NUMBER: Poly-EE/EP-76-023
7. AUTHOR(s): M. L. Shooman, H. Ruston	8. CONTRACT OR GRANT NUMBER(s): F30602-74-C-0294
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Polytechnic Institute of New York/Department of Electrical Engineering & Electrophysics, 333 Jay Street, Brooklyn NY 11201	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55500806
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441	12. REPORT DATE: March 1977
	13. NUMBER OF PAGES: 42
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Same	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT *(of this Report)*
Approved for public release; distribution unlimited.

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

18. SUPPLEMENTARY NOTES
RADC Project Engineer:
Captain Alan N. Sukert (ISIS)

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

Software Modeling	Computer Language Complexity
Software Testing	Programming Techniques
Software Physics	Software Micro Reliability Models
Software Complexity	Software Reliability

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*
During the period 1 July 1976 to 31 December 1976, Polytechnic Institute of New York ~~conducted~~ research under RADC Contract F30602-74-C-0294, in the area of software reliability. This report presents the progress of this research. Subjects of continuing investigation are as follows:

Work in progress includes development of a micro reliability model, incorporating representative features of the internal program structure, involving path (module) traversed frequencies and times and path failure

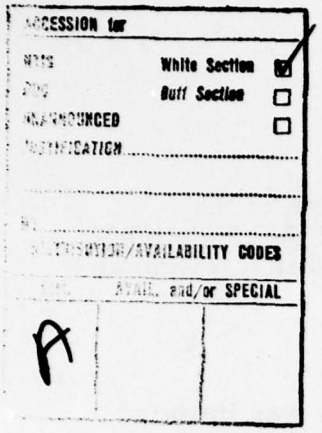
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

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

408 717

probabilities parameters; extension of software physics formulas to define a measure of complexity; planning and execution of small scale tests for gathering of parameters to verify modeling techniques; continuation of work on automatic and modular techniques for constructing low-cost, low-error content application programs; measurement of program length and complexity based on the application statistical natural language theory - (Zipf's Laws on work probabilities); algorithms for the enumeration of feasible program test paths; and development of driver programs for testing every program path at least once.



SECTION I

INTRODUCTION

This interim report summarizes the effort expended from July 1,1976 to December 31, 1976 on RADC Contract No. F-30602-74-C-0294. The major topics investigated and participating personnel during this period are listed in Section II. Summaries of these topics appear in Section III. Section IV indicates the planned direction of the research for the immediate future. Section V reports on the professional activities of the staff during the reporting period.

SECTION II

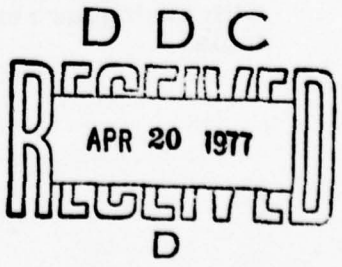
PERSONNEL AND WORK AREAS

The following personnel participated in the research activities during this reporting period

M. Shooman
H. Ruston

D. Baggi
E. Berlinger
A. Laemmel
E. Lipshitz
C. Marshall
S. Natarajan
G. Popkin
B. Rudner

and worked in the following areas:



1. Shooman and Nataranjan:

The macro model previously developed for the estimation of the number of errors has been extended. The extension involved the modeling of errors generated during debugging, and has been described in a technical report released in August 1976.

2. Shooman:

A new approach to the estimation of software reliability was initiated during this period. The approach centers on a micro reliability model. Such a model incorporates representative features of the internal program structure. Specific parameters of the model are the path (module) traversal frequencies and times, with the path failure probabilities.

3. Ruston and Berlinger:

Work has begun on extensions of software physics formulas to define a measure of complexity. The automation of gathering of statistics has been initiated for the validation of the extended formulas.

4. B. Rudner:

The seeding/tagging estimate formulas have been developed and are described in a technical report (released in November 1976). Several plans for a small scale experimentation to obtain experience with the method are being considered.

5. Ruston and Shooman:

Planning and execution of small scale tests for gathering of parameters for: the seeding/tagging estimates, the micro-models, and the extended software physics formulas.

6. Lipshitz and Shooman:

Continuation of work on automatic and modular techniques for the construction of low-cost, low-error content application programs.

7. Shooman and Laemmel:

A new approach to measurement of program length and complexity has been undertaken. This approach is based upon the application of statistical natural language theory. The specific theory exploited is the work by Zipf on word probabilities (in the 1930's), which has been shown to apply to program operands and operators. The results are presently being compared with analogous results achieved through the use of software physics formulas.

8. Popkin and Shooman:

Work is continuing on algorithms for the enumeration of feasible program test path.

9. Baggi and Shooman:

Continuation of the effort on driver programs. The driver programs developed so far are limited to programs with no loops. This work is being generalized to apply to all loopless programs. Attention is focused on eliminating this loopless restriction.

SECTION III

SUMMARY OF PROGRESS

In this section we describe briefly the work performed. Upon the completion of each task, a complete technical report will be issued. Several technical reports which document either the completed or continued research are in various stages of preparation.

3.1 Upper Bounds on the Number of Tests Needed to Verify a Computer Program

by Gary S. Popkin

3.1.1 Introduction

In an earlier report [1] the upper and lower bounds on the minimum number of program test cases were discussed, with application to flowcharts containing two-way decisions. The conditions for reaching the lower and upper bounds were discussed, and examples given. In this work, these ideas are extended to flowcharts containing three-way (e.g., $A < B$, $A = B$, $A > B$) and multi-way decisions.

3.1.2 Upper and lower bounds on the number of tests needed to verify a program

Each of the two flowcharts in Figure 1 contains four three-way decisions. The numbering of the segments, with two segment numbers on some of the flow lines, indicates that the decisions are three-way. In Figure 1(a), the methods of [2] would yield a maximum incomparable set size (and hence a lower bound on the minimum number of tests) of 3. It will be shown below how the upper bound may be computed, and how the flowchart contents can be inserted to raise the minimum number of tests required to approach the upper bound.

In Figure 1(b), the methods of [2] yield 9 as the size of the maximum incomparable set, and the lower bound on the minimum number of tests. 9 is also the upper bound, for no flowchart contents can raise the minimum number of required tests above 9.

Minimum number of tests for charts with three-way deciders

In a loopless flowchart with three-way decisions, the upper bound on the minimum number of tests needed to pass through each segment at least once is given by

$$u = 2d + 1$$

where d is the number of deciders in the flowchart.

Proof: Consider a flowchart with no deciders. Such a flowchart consists of one segment and requires one test. Each three-way decider added to the flowchart can require at most two additional tests, so $u = 2d + 1$.

Minimum number of tests for charts with multi-deciders

In a flowchart where the deciders may have any different numbers of outcomes, the upper bound on the minimum number of tests needed to pass through each segment at least once is given by

$$u = \sum_{i=1}^{n} w_i - n + 1$$

where w_i is the number of outcomes of decider i, and n is the number of deciders in the flowchart.

Proof: Consider a flowchart with no deciders. It consists of one segment and requires one test. Each decider i can require at most $w_i - 1$ additional tests, so

$$u = \sum_{i=1}^{n} (w_i - 1) + 1$$

$$= \sum_{i=1}^{n} w_i - n + 1$$

as asserted.

Figure 2 portrays the flowchart of Figure 1(a) with contents. The minimum number of tests needed to pass through each segment at least once is now 7. The computed upper bound for the flowchart is 9.

In the deciders in Figure 2, the segment numbers have the following meanings:

Segment No.	Outcome
1	P = 8
2	P > 8
4	P = 5
5	P > 5
7	P = 2
8	P > 2
10	P = 0
11	P < 0

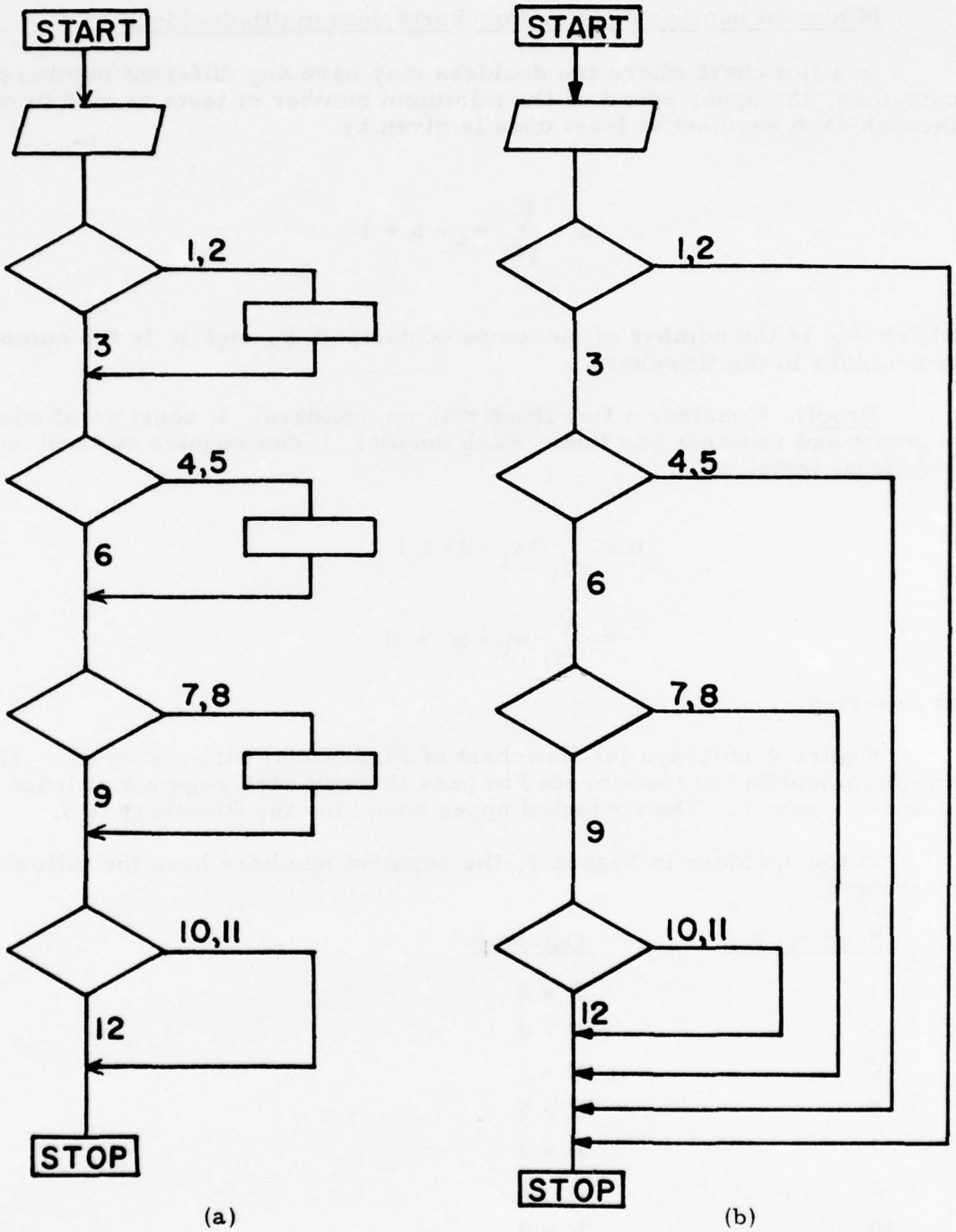


FIGURE 1

(a) A Flowchart with the Maximum Incomparable Set of Size 3

(b) A Flowchart with the Maximum Incomparable Set of Size 9

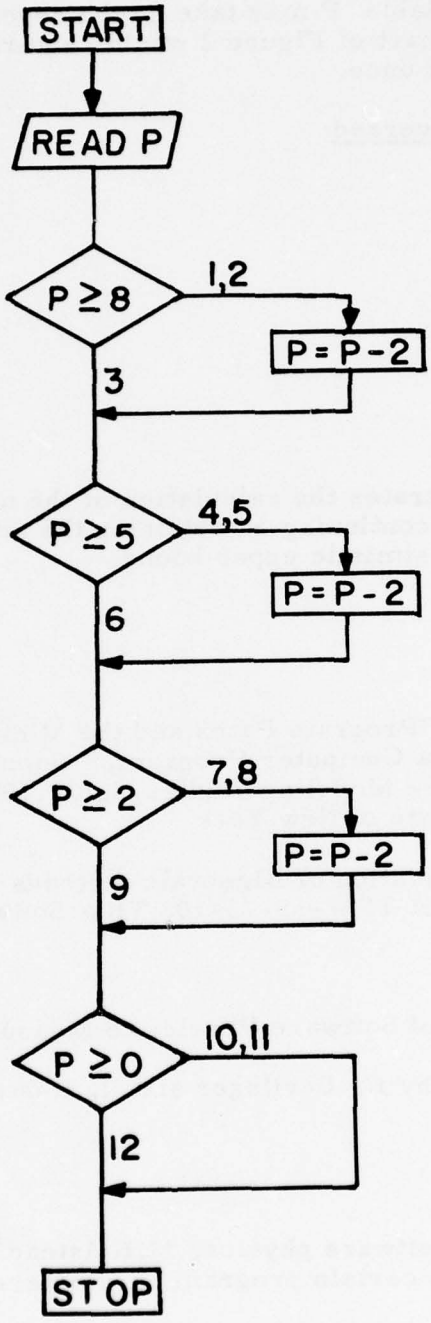


FIGURE 2

A Flowchart with Contents

If the input variable P may take on the seven values 1, 2, 4, 5, 7, 8, and 9, then the flowchart of Figure 2 would require seven tests to traverse each segment at least once.

P	Path traversed
1	3-6-9-11
2	3-6-7-10
4	3-6-8-12
5	3-4-8-12
7	3-5-8-12
8	1-5-8-12
9	2-5-8-12

The above illustrates the calculation of the upper bounds on the number of tests. Work is continuing on obtaining the actual number of tests rather than just a pessimistic upper bound.

1. Gary S. Popkin, "Program Paths and the Minimum Number of Tests Needed to Verify a Computer Program," Summary of Technical Progress, Software Modeling Studies, July 1, 1975-December 31, 1975, Polytechnic Institute of New York

2. M. Lipow, "Application of Algebraic Methods to Computer Program Analysis," Report TRW-55-73-10, TRW Software Series May 1973.

3.2 Extensions of Software Physics to Measures of Complexity

by E. Berlinger and H. Ruston

3.2.1 Introduction

In his work on software physics, M.Halstead[1] introduced a measure of complexity based upon certain program parameters. With the parameters:

N_1 = number of distinct operators

N_2 = number of distinct operands

a measure is

$$V = N_1 \log_2 N_1 + N_2 \log N_2$$

which he calls the program volume. Additional definitions and theorems are also given or conjectured. These relate the effort in programming, the total programming time, and estimate the program length.

Mathematically, the formulas are empirical and the proofs heuristic. But the results verify well with experimental data. The present effort attempts to improve on the Halstead scheme.

3.2.2 Outline of Current Work

One of the principal purposes of the work is to refine the definition of program volume to make it mathematically more sound, and also to include frequency of usage of the various program constructs. If we define:

f_1 = frequency of usage of the i^{th} operator

p_i = probability of usage of the i^{th} operator

f_j = frequency of usage of the variable whose rank is j

p_j = probability of usage of the variable whose rank is j

we can then define a measure of complexity as

$$-\sum f_i \log_2 p_i - \sum f_j \log_2 p_j.$$

There is strong justification for this definition from an information theory point of view. This measure should correlate well with the number of bugs in a program. If so, then the number of bugs can be predicted from an initial version of the program.

Work is currently focused on automating the process for gathering the statistics necessary to obtain the p_i and p_j. To this end, the operating system of an IBM 370/125 is being modified to copy all error-free FORTRAN student programs onto a tape. Student programs collected over a full semester will then yield the necessary probabilities.

Statistics on errors will also be collected and automatically copied onto a second tape. Specifically, the FORTRAN error numbers will be obtained from the output queue before printing. These will give the syntax and run-time errors. To obtain a count of logical errors, a count of the total number of times a program is run is being kept. It is assumed that the number of logical errors is one less than the number of runs which yield no syntax or run-time errors. This may be an underestimate but

since the programs being used are of first year programming students, it is reasonable to assume that they only find one logical error at a time.

To obtain the frequency counts, the tape containing the error-free runs will be run through a program supplied by Professor Halstead and Mr. Ottenstein of Purdue, which analyzes FORTRAN programs and yields the frequencies automatically. All programs on both tapes will be sufficiently identified so that the bugs can be correlated with the frequency counts.

Obtaining the probabilities p_i and p_j is a secondary purpose of the project and will supplement some statistics obtained previously by D. Knuth.

3.2.3 Tests for Obeyance of Zipf's law

It is expected that the probabilities p_i and p_j will also obey Zipf's law, either in its pure form (i.e., $p_r = \frac{c}{r}$, where p_r is the probability of the operator or operand whose rank is r), or in one of its modified forms (e.g., $p_r = \frac{c}{(r+a)^n}$). If the frequencies also follow a Zipf's law, it may be possible to get a criterion for program length. This, however, remains to be seen.

1. M. Halstead, "Software Physics, Basic Principles", IBM Research Report, RJ1582 IBM Research, Yorktown Heights, N.Y., May 1975.

3.3 Complexities of Natural and Computer Languages

by M. L. Shooman and A. Laemmel

3.3.1 Introduction

There is a great need for theoretical models which describe programs and allow us to quantitatively estimate complexity, running time, storage requirements, and development time. In addition to serving as an estimate during the initial design period, they can be refined as the program develops and used as a management and analysis tool. They can also be used to compare initial design approaches, programming styles, different algorithms, etc.. Early work on such a theory has been initiated[1]. This work discusses the lingiustic theory (Zipf's law's), extends these to programming languages, and develops equations for program length based on these principles. This work is similar to that of Halstead which is commonly known as "Software Physics"[2].

There are many similarities between natural and computer languages, and we will make use of the analogies between nouns and verbs and operands

and operators. The similarities in structure and content of natural and computer language are further illustrated via the following thought problem. Suppose we take a programmer who understands a particular computer language and give him the complete listing, comments, and documentation for a computer program. We instruct him to study the computer program until he understands it and then produce a report written in good English containing paragraphs, complete sentences, and algorithms written as a sequence of steps in good English without mathematical notation. In principle, the report and the computer program would be equivalent.

3.3.2 Zipf's law

Before we discuss Zipf's law it is convenient to introduce a few terms in dealing with natural language. We use the term token to refer to all the words of the written or spoken sample. The term type is used to refer to the vocabulary of words in the sample. Much of our efforts will be centered on the counting of the number of times, n_r, particular types occur in a sample of n tokens containing t types. The most frequently occurring type will be assigned rank $r = 1$, the second most frequent type rank $r = 2$, and the least frequent type rank $r = t$. Thus

$$\sum_{r=1}^{t} n_r = n \quad . \tag{1}$$

The absolute frequency of occurrence for type r is n_r; however, the relative frequency of occurrence f_r is simply n_r/n.

Zipf studied the relationship between relative frequency of occurrence f_r and rank r for words from English, Chinese, and the Latin of Platus[3].

Careful study of Zipf's data and that of others shows that f_r vs. r plots as a straight line on log-log paper, with a unity slope, thus we arrive at the simple relationship called Zipf's law,

$$f_r \cdot r = c \tag{2a}$$

$$n_r = \frac{cn}{r} \tag{2b}$$

Inspection of Eq. (2a) yields the fact that the constant c can be interpreted as the relative frequency of the rank 1 word type (also the intercept with the $r = 1$ line).

3.3.3 Type Token Equation

If we sum both sides of Eq. (2b), we obtain using Eq. (1)

$$n = \sum_{r=1}^{t} n_r = cn \sum_{r=1}^{t} \frac{1}{r} \cdot \qquad (3)$$

The summation of the series $1/r$ is given by[4]

$$\sum_{r=1}^{t} \frac{1}{r} = 0.5772 + \ln t + \frac{1}{2t} - \frac{1}{12t(t+1)} \cdots \qquad (4)$$

Substitution from Eq. (4) into Eq. (3)(retaining only 2 terms for modest size t) yields

$$n \approx cn\,(0.5772 + \ln t) \qquad (5)$$

We can eliminate the constant c from Eq. (5) by considering the behavior of Eq. (2b) for the smallest rank which is where $r_{max} = t$(e.g. if there are 100 types, then the largest rank is obviously 100). In most cases the rarest type (largest rank) will occur only once, thus, $n_{r_{max}} = 1$. Substituting these values yields, $c = t/n$, which when combined with Eq. (5) gives

$$n = t(0.5772 + \ln t) \qquad (6)$$

3.3.4 Summary of Experimental Results

The results to date have shown that both operators, operands, and the sum of operators plus operands rectify fairly well with a slope of unity on log-log paper (i.e. they fit Zipf's law) for:

a. An 11 and a 27 line PL/1 program (55 and 222 tokens)

b. The MIKBUG machine language executive program for the M6800 microprocessor (322 tokens)

c. Operators in PDP-11 assembly language programs (1572 tokens)

d. Variable names in 3 PL/1 programs (320, 238, and 193 tokens)

3.3.5 Relationship to "Software Physics"

The initial motivation for the application of Zipf's law to computer languages came from a review of Halstead's[2] work on Software Physics. Early in his work he arrives at a formula for program length

$$L = N_1 \log_2 N_1 + N_2 \log_2 N_2 \tag{7}$$

where

$L \equiv$ Program length

$N_1 \equiv$ Number of distinct operator types

$N_2 \equiv$ Number of distinct operand types

In terms of our notation the analogous quantities are

$$t = N_1 + N_2 \tag{8}$$

$$n = L \tag{9}$$

Note that Eqs. (7) and (6) are of similar form. In Ref. 1 we compare the actual number of tokens (counted) with the number of tokens calculated using both Eqs. (6) and (7). The average error and average magnitude error are computed, and both equations yield good agreement (10-20%), between actual and calculated results.

3.3.6 Estimation of Program Length Early in Design

One method of initially estimating program length (token length*) is to estimate the number of tokens. We assume the analyst initially has a complete description of the problem and that a partial analysis and choice of key algorithms has been made. An elementary approach might be to estimate the token size by

(1) Estimating the number of operator types which will be used in the language by the assigned programmers.

(2) Estimate the number of input variables, output variables, intermediate variables, and constants need.

(3) Sum the estimates of step (1) and (2) and substitute in Eq. (6).

---

* In addition one must add other classes of statements and programming elements such as: comments, declares, certain assembler directives, etc.

Clearly we might consult past programs written by the assigned programmers for data on number of operator types; also, if a large program will be stated in a comprehensive specification document, written in English or in a specific alien language. This should name input and output variables; thus our estimate will mainly deal with predicting the number of intermediate variables and constants.

1. A. Laemmel and M. Shooman, "Statistical (Natural) Language Theory and Computer Programs", Poly EE/EP Report, January 1977.

2. M. Halstead, "Software Physics, Basic Principles" IBM Research Report, RJ1582, IBM Research, Yorktown Heights, N. Y., May 1975.

3. George K. Zipf, "The Psycho-Biology of Language: An Introduction to Dynamic Philology" M.I.T. Press, 1965. (First Houghton Miffin Edition, 1935).

4. L. Jolley, "Summation of Series," p. 36, n. 200, and p. 14, n. 70, Dover Publications, New York, 1961. (Note the constant 0.5772 is called Eulers constant, see "Differential and Integral Calculus," R. Courant, vol. 1, Interscience Publishers, New York, 1951, p. 381, 420).

3.4 Experimental Verification of Debugging Models

by D. L. Baggi

3.4.1 Introduction

The object of this study is the implementation of a so-called driver program which, given a program to test, will force the traversal through all its possible paths. The advantage of such a procedure is obvious for programs with several branches and decision points; they are usually debugged by laborious construction of a data set, which hopefully would cause exploration of all paths. The method described here forces exhaustive testing of all possible paths, with no need for the design of a data set.

3.4.2 Initial Drivers

The first effort in the definition of the driver program consisted of the implementation of a program capable indeed of traversing all paths of a given program. This program iteratively substitutes, in place of the condition in a PL/1 IF-statement, a value of zero or one, alternatively. With this done concurrently for all conditional statements, eventually one traverses once all possible paths. It was realized (already by Shooman, in his paper "Analytical Models for Software Testing") that the resulting number of

paths of 2^n for conditional branches, is merely an upper bound for all possible paths, where the real number lies in fact, between $n+1$ and 2^n. Hence the above scheme wastes execution time, which increases exponentially (for example, a program with fourteen paths may require 8142 runs, of which 8128 are meaningless!). Thus an algorithm had to be designed to consider the possible paths only.

3.4.3 Present Drivers

The present algorithm scans a PL/1 program. It searches for keywords such as IF, THEN, ELSE, DO and END, while constructing a regular expressions of zeroes and ones. Resolution of this regular expression yields a set of binary integers, which represent the status of the conditional expressions during each of the runs through all possible paths. In fact, each bit of such integers represents the value of the next conditional expressions met during execution, hence forcing a zero or one - branch accordingly. Since those integers were derived from the very structure of the algorithm of the program, thev represent indeed each path; as an extra bonus, their total number is the number of possible paths. Hence the algorithm enumerates each path, uniquely describing it in terms of its branches, and also counts them.

The algorithm works as follows:

- each expression is binary; it contains two terms separated by a + sign

- the terms can be only 1, 0, or 1 or 0 concatenated with a binary expression

Scanning rules:

a. each IF opens a left parenthesis, (;

b. each THEN corresponds to a 1 ;

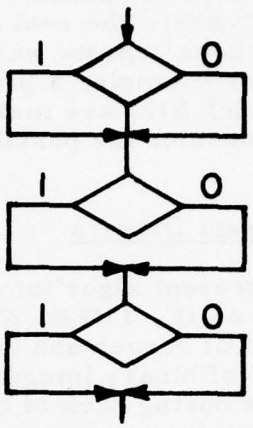
c. each ELSE corresponds to a 0 ;

d. each well completed binary expression, with both terms completed, compels closing with right parentheses at its level.

Note: The ELSE clause is assumed, for convenience, to be always present.

3.4.4 Examples:

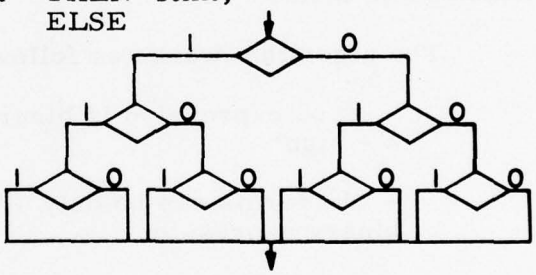
```
IF cond THEN  stmt;
          ELSE  stmt;
IF cond THEN  stmt;
          ELSE  stmt;
IF cond THEN  stmt;
          ELSE  stmt;
```



Result:

(1+0)(1+0)(1+0)
or the eight paths
111, 011, 101, 001, 220, 010, 100, 000

```
IF cond THEN IF  cond THEN IF  cond  THEN  stmt;
                                     ELSE  stmt;
                          ELSE IF  cond  THEN  stmt;
       ELSE IF  cond THEN IF  cond  THEN  stmt;
                          ELSE IF  cond  THEN  stmt;
                                     ELSE
```



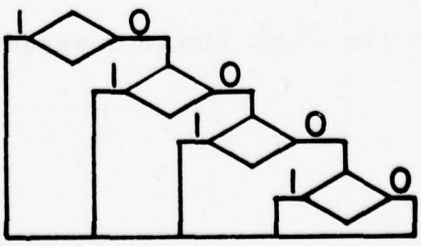
Result:

(1(1(1+0)+0(1+0))+0(1(1+0)+0(1+0)))
which gives the eight paths
111, 110, 101, 100, 011, 010, 001, 000

```
IF  cond  THEN  stmt;
        ELSE IF  cond  THEN  stmt;
                          ELSE IF  cond  THEN  stmt;
                                     ELSE  stmt;
```

yields:

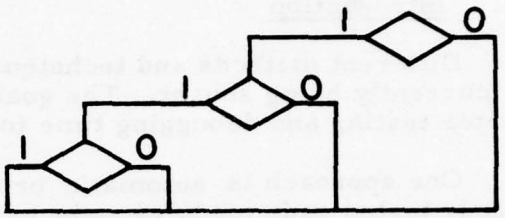
(1+0(1+0(1+0(1+0))))
or
1, 01, 001, 0001, 0000



```
IF cond  THEN DO; IF  cond THEN DO; IF cond  THEN stmt;
                                                ELSE  stmt;
                                  END;
                             ELSE  stmt;
                 END;
          ELSE  stmt;
```

gives:

(1(1(1+0)+0)+0)
i.e.,
111, 110, 10, 0



Because of the recursive nature of the algorithm, it has been implemented in the language LISP. It could eventally be translated in PL/1. In the meantime, however, the greatest concern still lies in making sure that such an approach, and such an algorithm, works at all; hence the choice of LISP for fast implementation. No attention was given so far to repetitive DO-groups, which were appropriately handled by the initial crude algorithm. This is because extension of this algorithm to such loops is conceptually very trivial, and it can be proved that the system would perform equally well: but, in the meantime, such an extension looks very time consuming and would by no means add any contribution to the theory of this debugging model.

3.4.4 Direction for Further Work

The long range idea is to eventually come up with a complete PL/1 package, capable of exploring all paths of a program, paying attention to IF-statements, DO-loops, etc. In the meantime, however, the attention is given to producing a tentative system capable of showing that such a project is indeed possible. To this end, the following set of programs is under construction:

1) a PL/1 program which reads in the object program (the one under debugging) and translates it in LISP-compatible notation; this is saved on a file

2) a LISP program which scans the object program and constructs the regular expressions, with results saved on a second file.

3) a PL/1 driver program which reads the results of these expressions and forces execution of the object program through all its possible paths.

3.5 Automatic Programming Techniques

by E. Lipschitz

3.5.1 Introduction

Different methods and techniques for writing a better software package are currently being sought. The goal is to write programs which require a shorter testing and debugging time to achieve a certain degree of reliability.

One approach is automatic programming. The use of pre-written and already tested code modules reduces the number of bugs.

3.5.2 The Program "AUTO-PROGRAMMING"

The working hypothesis is that there exists a high degree of commonality among commercial applications which can be exploited to automate the production of code, once processing and output specifications are defined.

"AUTO-PROGRAMMING" is divided into two parts -- "Flow" and "Auto", both of which are interactive on-line programs that, by communicating with the user, generate his programs.

A. The Program "Flow"

"Flow" receives the information about the flow-chart of a program from the user and generates it. "Flow" recognizes only four different types of blocks, which are sufficient to generate any flow-chart. They are:

Type #1; Control block: A conditional decision block, similar to the statement If (_____). Go to (_____).

Type #2; Functional block. This block will perform a complete task selected from those in the computer library.

Type #3; Stop block. This block indicates the end of a path; i.e., Stop statement.

Type #4; User's code block. The user inserts the code he wants into this block. This feature is used whenever the library does not include programs for the needed task.

Upon completing the flow-chart, control passes to "Auto". "Auto" will generate the code for blocks type #2 and #3, while the user will generate the code for blocks type #1 and #4.

B. The Program "Auto"

A collection of code modules can be stored as a system library. The desired program can be achieved by concatenating different code modules from the system library with code generated by the user.

The user will specify to "Auto" what he would like to do, and "Auto" will advise him which methods are available for the solution, as well as their characteristics. The user then will choose the method he prefers, and "Auto" will generate the needed code.

The library currently contains the following modules.

1. Linear Search
2. Binary Search
3. Interchange Sort
4. Shell Sort
5. Sin(x) $\qquad 0 \leq x \leq 2\pi$

$$\text{Sin}(x) = \sum_{n=0}^{m} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

6. Cos(x) $\qquad -2\pi \leq x \leq 2\pi$

$$\text{Cos}(x) = \sum_{n=0}^{m} \frac{(-1)^n x^{2n}}{(2n)!}$$

7. $$\text{Ln}(x) = 2 \sum_{n=0}^{m} \left(\frac{1}{2n+1}\right)\left(\frac{x-1}{x+1}\right)^{2n+1}$$

8. $$\text{Exp}(x) = \sum_{n=0}^{m} \frac{x^n}{n!}$$

9. Arctan(x) $\qquad$ for $|x| \leq |$

$$\tan^{-1}(x) = \sum_{n=0}^{m} \frac{(-1)^n x^{2n+1}}{2n+1}$$

10. Bessel Function of the First Kind and Zero Order.

$$I_0(x) = \sum_{n=0}^{m} \frac{\left(-\frac{x^2}{4}\right)^n}{(n!)^2}$$

11. Bessel Function of the First Kind, Orders 0, 1 and 2

$$J_0(x) = \sum_{n=0}^{m} \frac{\left(-\frac{x^2}{4}\right)^n}{(n!)^2}$$

$$\frac{2}{x} J_1(x) = \sum_{n=0}^{m} \frac{\left(-\frac{x^2}{4}\right)^n}{(n!)^2 \cdot (n+1)}$$

$$J_2(x) = \frac{2}{x} J_1(x) - J_0(x)$$

12. Modified Bessel Function of the First Kind, Orders 0, 1 and 2

$$I_0(x) = \sum_{n=0}^{m} \frac{\left(\frac{x^2}{4}\right)^n}{(n!)^2}$$

$$\frac{2}{x} I_1(x) = \sum_{n=0}^{m} \frac{\left(\frac{x^2}{4}\right)^n}{(n!)^2 \cdot (n+1)}$$

$$I_2(x) = -\frac{2}{x} I_1(x) + I_0(x)$$

13. Error Function

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{m} \frac{(-1)^n x^{2n+1}}{(n!)(2n+1)}$$

14. Fresnel Integral C(x)

$$C(x) = \int_0^x \cos\left(\frac{\pi}{2} t^2\right) dt = \sum_{n=0}^{m} \frac{(-1)^n \left(\frac{\pi}{2}\right)^{2n} x^{4n+1}}{(2n!)(4n+1)}$$

15. Fresnel Integral $C_2(x)$

$$C_2(x) = \frac{1}{2\pi} \int_0^x \frac{\cos t}{\sqrt{t}}\, dt = \frac{1}{\sqrt{2\pi}} \sum_{n=0}^{m} \frac{(-1)^n x^{2n+1/2}}{(2n!)(2n+1/2)}$$

16. Fresnel Integral $S(x)$

$$S(x) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right) dt = \frac{\pi}{2} \sum_{n=0}^{m} \frac{(-1)^n \left(\frac{\pi}{2}\right)^{2n} x^{4n+3}}{[(2n+1)!](4n+3)}$$

17. Sine Integral $S_i(x)$

$$S_i(x) = \int_0^x \frac{\sin t}{t}\, dt = \sum_{n=0}^{m} \frac{(-1)^n x^{2n+1}}{[(2n+1)!](2n+1)}$$

18. Cosine Integral $Cin(x)$

$$Cin(x) = \int_0^x \frac{(1-\cos t)}{t}\, dt = - \sum_{n=1}^{m} \frac{(-1)^n x^{2n}}{[(2n)!](2n)}$$

19. Dilogarithm $f(x)$

$$f(x) = - \int_1^x \frac{\mathrm{Ln}\, t}{(t-1)}\, dt = \sum_{n=0}^{m} \frac{(-1)^n (x-1)^n}{n^2}$$

Note: m is so chosen that the magnitude of the m^{th} term of the power series is less than or equal to 10^{-6}, while the magnitude of the $(m-1)^{th}$ term is greater than 10^{-6}.

3.5.3 Conclusion

The development of "AUTO-PROGRAMMING" will continue to concentrate mainly on increasing the size of the library. More mathematical programs, as well as some utility programs for data manipulation, will be developed in the near future.

3.6 Small Scale Tests

by H. Ruston and M. L. Shooman

3.6.1 Introduction

In order to verify the theoretical models four programs have been written by student programmers and careful records were kept on their debugging experiences. We will describe the assigned programs and the error reporting form. This data is presently being reduced and the resulting conclusions will be described in a following report.

3.6.2 The Programmers

The programmers were undergraduate students of sophomore-junior standing, with high interest and ability in programming topics. Because of this selectivity we believe their product to be likely the one equivalent to the one of programmers with intermediate experience.

Consequently, we consider the obtained test data to be representative of normal practice.

3.6.3 The Instructions

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 analyses and coding times recorded.

2. The resulting 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 print-outs were to be saved and numbered.

3. The programs were then presented to us (i.e., M. Shooman and H. Ruston). We planned to ask 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. If a programmer reached a blind alley he had to consult with us. He could not ask for other help, or abort the program

5. The program had to be constructed with the following constraints:

   a. To be structured

   b. To contain no impurities (as listed in Halstead[1])

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

   d. No module to exceed 50 lines

3.6.4 The Four Programs

Three small problems (problems 1, 2, and 3) and one medium size problem were generated for the small scale tests. The initial write-ups of the problems for the desired four programs follow.

Problem # 1

Minimum Salary Payroll Adjustment

1. Statement of the Payroll Adjustment

   Glen Cove University which employs 200 faculty members has just signed a non-faculty union contract. All salaries of $16,000 or higher are to remain unchanged. Any faculty member who earns less than $16,000 per year is to receive a pay raise according to the following formula: He will receive 100 per year additional for each dependent (including himself), plus 50 per year for each year of employment. In no case may his new salary exceed $16,000 per year.

   The personnel data on all faculty members is stored on magnetic tape in the Business Office and includes present annual salary, number of dependents, date of hire and other information. The problem is to write a program which computes and prints out the list of faculty members along with their old and new salary.

   Assume that the Business Office will give you a set of cards with the data for each person on one card. You should create your own test data; however, final testing of your program will be done on the actual card deck. Assume the following arrays will accept the input data in your program:

   NAME (200): contains a name of up to 30 characters

   DEPEND (200): contains number of dependents in two decimal digits

DHIRE (200): contains a string of 10 characters, two digits for month, a blank, two digits for day, a blank, and four digits for year

PRESS SAL (200): contains 5 digits with yearly salary in rounded dollars

2. Approaches (write two programs, using each approach):

   (a) Search list of names for those making less than $16,000 per year, compute new salary, check $16,000 limit, store new salary, print output

   (b) Sort list by salary from lowest to highest, stop when $16,000 is exceeded, compute new salary, check $16,000 limit, store new salary, print output

3. Language and Computer: PLAGO on Poly 360/65

Problem #2

Finding the Roots of a Cubic Equation

1. Statement of Problem

   The polynomial equation $a_3x^3 + a_2x^2 + a_1x + a_0 = 0$ is to be solved for its three roots. A general solution is desired which will work for any finite real values of a_3, a_2, a_1 and a_0. The values of a_3, a_2, a_1, and a_0 are to be acquired as floating (single precision) input data at the beginning of each run. Write your program with a loop so it reads any number of data cards and terminates on last data card. Make up your own test data; however, it will be tested finally with supplied data cards.

2. Approaches

   (a) The general formula (i.e., Cardano's formula) for the solution of a cubic (see attachment A) is to be used to compute the roots.

   (b) An iterative solution for a single real root is to be obtained. Once the real root is removed, the quadratic formula is to be used to solve for the other two roots.

3. Language and Computer: PLAGO on Poly 360/65

Problem # 3

Manipulation of a File of Research Reports

1. Statement of the Problem

At present the Reliability, Safety, and Software Engineering Group at the Polytechnic has about 1000 reports, papers, books, journal proceedings in its library. Each item is entered on an index card in a file box. It is anticipated that the library may eventually grow to 10,000 items in the future. In the near future, two punched cards will be created for each of the items in the library and we wish to create a program to perform various searches, sorts, and listings.

Assume that each punched card contains 4 character fields. The first field is 30 characters wide and contains the author(s) name(s). The second field is 50 characters wide and contains the full or abbreviated title. (No important words in the title are to be abbreviated.) The second card contains field 3 which is 50 characters wide and contains key words (or abbreviated key words) in the item. The last field is 30 characters wide and contains the source (journal, issue, book publisher, proceedings, etc.) of the item.

The program must be able to perform the following tasks, and the selection (and possibly sequence) of tasks to be performed must be controlled by the first data card which will serve as a program control card. Provide a means of performing tasks on the same run.

(1) Read in a variable length stack of item cards and store them;

(2) Alphabetize the data by first author;

(3) Print out the list of items;

(4) Create a list of key words (from field 3), eliminate duplicates, alphabetize, and print out;

(5) Search the author field for a given author's name, and print out the list of items he has written;

(6) Search the key word field for items which contain the "intersection" (AND) of one, two, or three inputed key words;

(7) Provide the same search facility as (6) on words in the title field.

Programmer should create his own test cards, and final testing will be performed by a supplied deck of item cards.

(8) Approaches: Programmer should provide his own approaches.

(9) Language and computer: PLAGO on 360/65

Problem #4

Specifications for Ballot Counting Procedure

Terms: An election consists of sets of ballots to elect persons for various committees. Each set of ballots is called a committee election. There may be up to 10 committee elections for an election.

All ballots for a particular committee have the name of the committee punched in columns 61-80 and contain the names of the candidates to that committee. There are up to 25 nominees for each committee.

When a person votes for a candidate, an "11" punch (i.e., a minus sign) is punched in the ballot in the field consisting of columns 21-45 corresponding to candidates 1-25. Such a punch is a mark. A particular ballot may have more than one mark since a particular committee may have more than one vacant position (i.e., there may be more than one vote allowed to each voter on each ballot).

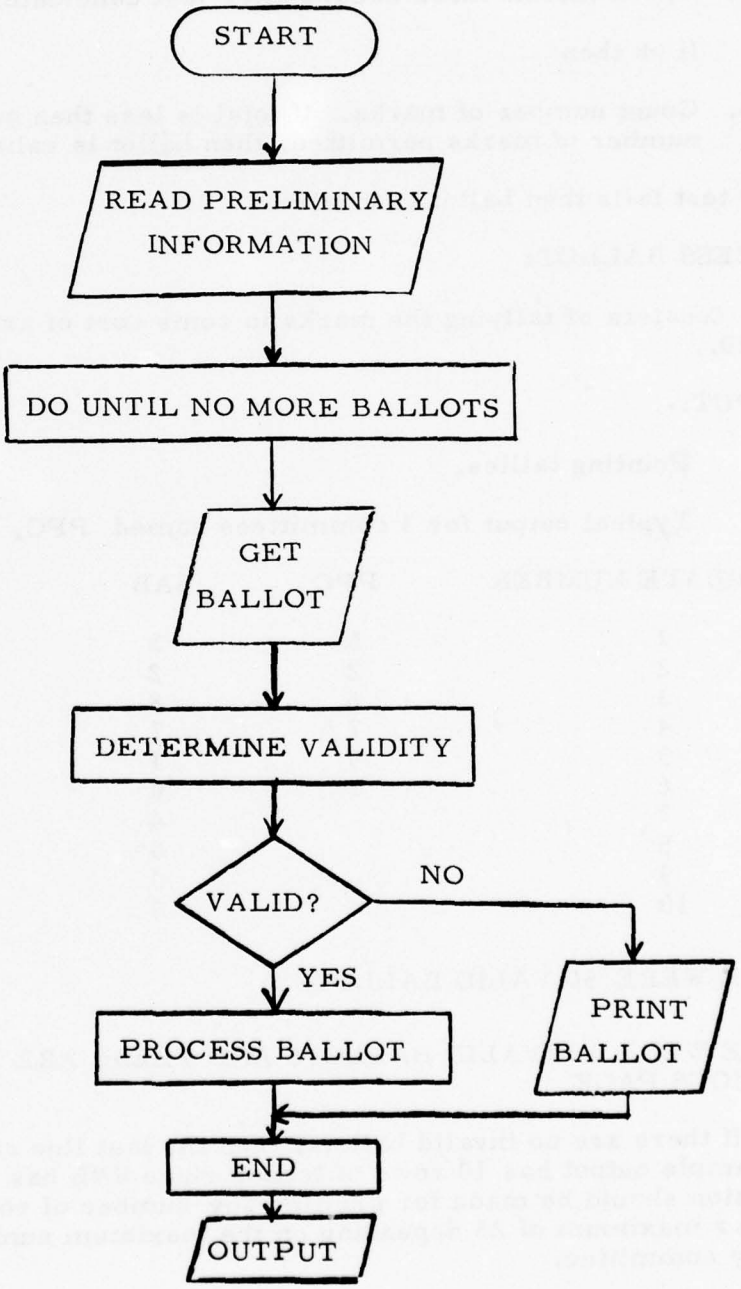
An election package consists of one ballot for each committee. Each eligible voter receives one and only one election package which he marks and returns for counting.

Specifications: Prior to counting the ballots, the program must read certain preliminary information concerning the election.

For each committee election the program must be informed as to:

1. the name of the committee
2. the number of candidates
3. the number of marks permitted (i.e., if "vote for three" then three marks are allowed).

A rough flow chart is:



DETERMINE VALIDITY:

1. Check election name (cols. 61-80). If name is found then
2. Check all marks (cols. 21-45) to see if all are '-' or ' '. If ok then
3. Check that no mark occurs after last candidate.

   If ok then

4. Count number of marks. If total is less than or equal to the number of marks permitted, then ballot is valid

If any test fails then ballot is invalid.

PROCESS BALLOT:

Consists of tallying the marks in some sort of array, probably 25 x 10.

OUTPUT:

Printing tallies.

Typical output for 3 committees named PPC, SAB, TENURE

CANDIDATE NUMBER	PPC	SAB	TENURE
1	5	3	9
2	2	2	2
3	6	8	3
4	7	7	5
5	9	1	8
6	1	0	7
7		4	2
8		0	5
9		0	
10		8	

THERE WERE 50 VALID BALLOTS

THERE WERE 4 INVALID BALLOTS AND THESE ARE REPRINTED ON PREVIOUS PAGE

If there are no invalid ballots, then the last line can be left unprinted. The sample output has 10 rows of tallies since SAB has 10 candidates. Provision should be made for printing any number of rows from a minimum of 2 to a maximum of 25 depending on the maximum number of candidates for any committee.

3.6.5 The Reporting Form

After several revisions the resulting reporting form has been selected.

POLYTECHNIC INSTITUTE OF NEW YORK

Department of EE/EP

Division of Computer Science

Safety, Reliability and Software Engineering Group

RADC Contract F30602-74-C-0294 (Softy)

INDIVIDUAL ERROR REPORTING FORM

(This must be completed for each non-syntax error)

1. Identification

   (a) Programmer's Name _____

   (b) Program Title _____

   (c) Date _____

   (d) Form Number (for this program)_____

   (e) Description of the error (be precise) _____

   _____

   _____

   _____

   _____

   (f) Description of the correction (be precise) _____

   _____

   _____

   _____

   _____

2. Means of Detection -✓only for Corrections (not New Reqs.)
   -More than one category may be ✓'ed

   ☐ a. Hand Processing ☐ d. Interrupt Error (Code____)

   ☐ b. Personal Communication ☐ e. Incorrect Output or Result

   ☐ c. Infinite Loop ☐ f. Missing Output

   ☐ g. Other - Explain

3. Effort to Diagnose the Error - Do not include effort spent in initial detection

   a. No. of Runs to Diagnose___Elapsed Computer Time (Minutes)___

   b. Working Time to Diagnose Hours___

4. Category of Change

SOFTWARE CHANGE REQUIRED

Nature of Change

- ☐ Documentation (Preface or Comments)
- ☐ Fix Instruction
- ☐ Change Constants
- ☐ Structural
- ☐ Algorithmic
- ☐ Other - Explain

Source of Bug

- ☐ Bug essentially unrelated to previous corrections (i.e., usual case of bug just discovered)
- ☐ Previous correction did not remove the believed error (i.e., improper or incomplete analysis)
- ☐ New Bug, introduced by a previous correction (i.e., bug generation through a correction).

Type of Bug

- ☐ Misinterpretation of Specifications
- ☐ Wrong Specifications
- ☐ Incomplete Specifications
- ☐ Incorrect Sequencing of Computations
- ☐ Incorrect Input Data (Type and Quantity)
- ☐ Incorrect Expressions
- ☐ Incorrect Declaration
- ☐ No Defense Against Invalid Data
- ☐ Operating System
- ☐ Support Software
- ☐ Card Mispunched
- ☐ Other - Explain

5. Difficulty of Correction

   a. No.of Runs to Correct___Elapsed Computer Time (Minutes)___

   b. Working Time to Debug: Days___Hours___

   c. No. of Cards: Changed___Added___Deleted___

6. Comments (Use Reverse Side and Additional Sheets If Necessary)

3.6.6 The Present Status and Planned Activities

Problems 1, 2 and 3 have been debugged by single programmers, that is, by just their authors. A set of test data has been selected for problem 1, and the several versions of the first program have been exercised with this test data successfully.

It is planned to perform the additional debugging with other programmers and to use the test data for the experimental small scale verification of our theoretical work.

1. M. Halstead, "Software Physics", Basic Principles IBM Research Report, RJ1582 IBM Research, Yorktown Heights, N.Y. May 1975.

3.7 Micro Reliability Models

by M. L. Shooman

3.7.1 Introduction

Many previous software reliability prediction models by this author[1] and others[2] have concentrated on the bulk (macro) aspects of a program. This work involves a newly developed micro model[3] which is based on program structure.

It is assumed that the program has been written in structured or modular form so that decomposition into its constituent parts is simple. Further, we assume that via analysis of the program the decomposition can be related to several paths or other functional structures within the program.

The model is constructed based upon the frequencies with which each of the j paths are run, (f_j), the running time of each path, (t_j), and the probability of error along each path, (q_j).

Several methods of calculating or measuring the f_j, t_j, and q_j parameters are suggested. In fact it is possible to use one technique (historical data) to produce crude estimates at the start of the design, and refine the estimates with more accurate values as the design progresses. Given the existence of such a model, we can consider the application of three important design techniques which are impossible with a macroscopic model:

(1) Apportionment of the software reliability (or mean time to failure) specification among the subsystems so each design team has their own goal to meet. The apportionment is obviously done so that the subsystem reliabilities combine to yield a system reliability which meets system specifications.

(2) If design proceeds either bottom up or top down, eventually there is a system integration phase where all parts are put together and tried out. The macroscopic models developed previously could not be applied before the system reached the integration stage. However, the new microscopic model proposed can be used to combine the results of the module development phases to predict a preliminary software reliability index before the system integration phase.

(3) The microscopic model is based upon measurements made on the software design. Such measurements and analyses performed on the software lead to a more disciplined design and provide insight into how the module performance relates to overall software system performance.

3.7.2 Micro Decomposition Model

The micro decomposition model which will be proposed in this section is based upon several assumptions. We first assume that the program has been designed using a structured or modular philosophy and as a result there emerges a natural structure of the program which can be described as consisting of a number of paths, cases, parts, modules, or subprograms. The decomposition focuses about this natural structure. In general we will primarily use the term paths from now on to designate the paths, cases, parts, modules, subprograms, or any other important substructure. We also assume that the majority of the paths are independent of each other. (One could probably tolerate some type of dependence in the model if it were limited.)

The decomposition model will be developed from the probabilistic viewpoint of relative-frequency. We will hypothesize a sequence of tests which either uncover a bug (failure) or run to completion without uncovering a bug (success). We begin our development of the model by defining the following variables and parameters:

$N \equiv$ The number of tests

$i \equiv$ The number of software paths (cases, parts, modules, etc.)

$t_i \equiv$ Time to run case i (if time is not deterministic we can substitute the mean value of t_i, i.e., $\bar{t}_i$).

$q_i \equiv$ Probability of error on each run of case i (The probability of no error $p_i = 1 - q_i$).

$f_i \equiv$ Frequency with which case i is run.

$n_f \equiv$ Total number of failures in N tests.

$H \equiv$ Total cumulative test time in hours.

Note that in the above set of definitions we have defined N as the number of tests. Thus, we are modeling actual or simulated operation by a succession of N tests (path traversals) of the system. We also assume the input data varies on each traversal. This is the reason why we have assigned a constant as the probability of encountering a bug on each run, q_i.

If there were no variation in input parameters, and three successive tests each traversed path j, then the probability of encountering an error on the first trial would be q_i. The conditional probability of encountering a bug on the second traversal of the same path with the same parameters is unity. Similarly, the probability on the same path with the same parameters the third time is also unity. Thus, the probability of a bug on three traversals of path j is $q_j \times 1 \times 1 = q_j$.

Since we have assumed a variation in parameters on each run in our model and each test is independent, then the probability of encountering one bug on three successive traversals of path j is given by the binomial distribution as [3]

$$P(1 \text{ error in three trials}) = \binom{3}{1} q_j^1 \left(1-q_i\right)^2 \qquad (1)$$

Similarly, the expected number of occurences in a probabilistic process governed by a binomial distribution is

$$\text{Number of Occurrences} = Nq \qquad (2)$$

where N is the number of trials and q the probability of occurrence.

3.7.3 Development of the Model

We can now compute the total number of failures n_f in N tests. The tests are distributed along each path such that Nf_1 tests traverse path 1, Nf_2 tests traverse path 2, etc. Thus, successive application of Eq. 2 to each of the i paths yields for the number of failures in N tests.

$$n_f = Nf_1q_1 + Nf_2q_2 + \ldots\ldots Nf_iq_i \qquad (3)$$

We can now compute the system probability of failure on any one test run, q_o, by taking the ratio of n_f/N as N approaches infinity

$$q_o \equiv \lim_{N \to \infty} \frac{n_f}{N} = \sum_{j=1}^{i} f_j q_j \qquad (4)$$

Similarly we can compute the system failure rate, z_o, by first computing the total number of test hours. First we compute the total number of

traversals of path i as Nf_i as was previously done. Out of these traversals the fraction p_i will be successful and will accumulate $N \times f_i \times p_i \times t_i$ hours of successful operation. If we assume that the time to failure distribution for the $Nf_i q_i$ traversals which result in failure is rectangular, then each trial which results in failure runs $t_i/2$ hours on the average before failure. Thus, the total test time accumulated in N runs is given by

$$H = Nf_1 p_1 t_1 + Nf_1 q_1 \frac{t_1}{2} + Nf_2 p_2 t_2 + Nf_2 q_2 \frac{t_2}{2}$$

$$+ \ldots . Nf_i p_i t_i + Nf_i q_i \frac{t_i}{2} = N \sum_{j=1}^{i} f_i t_i \left(p_i + \frac{q_i}{2}\right) \tag{5}$$

Substitution for $p_i = 1 - q_i$ in Eq. (5) and simplification yields

$$H = N \sum_{j=1}^{i} f_i t_i \left(1 - \frac{q_i}{2}\right) \tag{6}$$

we now compute the system failure rate z_o as

$$z_o = \lim_{N \to \infty} \frac{n_f}{H} \tag{7}$$

and substitution from Equations (3) and (6) into (7) yields in the limit

$$z_o = \frac{\sum_{j=1}^{i} f_j q_j}{\sum_{j=1}^{i} f_j \left(1 - \frac{q_j}{2}\right) t_j} \tag{8}$$

3.7.4 Special Cases

We now wish to examine equations (4) and (8) under special constraints. These are listed in Table 1. Note that the units of z_o are clearly seen from case 4 to be failures per hour, or just $hr.^{-1}$.

3.7.5 Measurement of Parameters

In order to implement the model developed in the previous sections we must develop numerical values for the sets of parameters f_j, q_j, and t_j.

Of course in keeping with the concept of structured programming and levels of structures within levels, one could merely state that we continue decomposition to lower levels until we end up with a new set $f'_{j'}$, $q'_{j'}$, and $t'_{j'}$ parameters at a lower level. Clearly, the answer to the question how we could measure or estimate our parameters at a higher level is also, by and large, an answer to how we would do the measurement at a lower level.

The parameter sets f_j and t_j are related to the structure, size, and complexity of the control structure and program modules. The determination of the f_j can be made by a study of the physical meaning of the paths and the distributions of input parameters which drive one along the program paths. If the program is complex, or there is really no information on input statistics, we can take one of two approaches. Assume f_j has a uniform distribution (see case 2 of Table 1) or insert counters in the various paths, and experimently determine the f_j. The experimental approach requires that the program be in reasonably good shape so that a simulated test program can be run. Clearly, if a counter c_j is placed in each path such that it registers one count for each path traversal, and we run N tests then $f_j = c_j/N$.

The set of t_j parameters can also be either calculated or measured. If the program is written in assembly, machine or microprogramming code, one can estimate quite closely the run time of a sequence of code by summing the operating times of each instruction. If the program is complex, one can write an analysis program to read the code and perform the time analysis to determine t_j. In the case of a higher level language, (FORTRAN, PL/1, COBOL, etc.) the analysis is more complex, because each statement may expand into one to say ten machine language statements. Several approaches are possible. First of all, one can obtain a core dump of the machine language program and proceed as has been described. Another alternative is to insert a block of higher level code inside a DO I = 1 to K loop. The loop is run for a particular value of K and the C.P.U. time of the computer recorded. The value of K is changed and another run and value of C.P.U. time is recorded. With about 3 values of C.P.U. time vs. K an accurate enough straight line or polynomial model of run time vs. K can be fixed to the data.* One can then use the formula to predict the run time of the actual code block by substituting the number of repetitions. (True value of K.) Of course if the program and a simulation is available one can merely run several test runs for each path, record the times and use average values for each path.

---

* It is necessary to take several measurements for two reasons. First of of all there is program overhead which may vary from run to run depending on the operating system. (Also there is DO loop overhead). Second, the recording of C.P.U. time is not accurate for short run times. To correct for DO loop overhead and also system overhead, one can perform the measurement with and without the code block in the loop and work with the difference.

TABLE 1. System Probability of Failure and Failure Rule for Special Cases

Constraints	$q_o \equiv \sum_{j=1}^{i} f_j q_j$	$z_o = \dfrac{q_o}{\sum_{j=1}^{i} f_j \left(1 - \frac{q_j}{2}\right) t_j}$
Case 1		
$q_j << 1$	$\sum_{j=1}^{i} f_j q_j$	$\dfrac{q_o}{\sum_{j=1}^{i} f_j t_j}$
Case 2		
$q_j << 1$		
$f_1 = f_2 = \ldots f_i = f = \frac{1}{i}$	$\frac{1}{i} \sum_{j=1}^{i} q_j$	$\dfrac{\sum_{j=1}^{i} q_j}{\sum_{j=1}^{i} t_j}$
Case 3		
$q_j << 1$		
$f_1 = f_2 = \ldots f_i = f = \frac{1}{i}$	q	$\dfrac{iq}{\sum_{j=1}^{i} t_j}$
$q_1 = q_2 = \ldots q$		
Case 4		
$q_j << 1$		
$f_1 = f_2 = \ldots f_i = f = \frac{1}{i}$		
$q_1 = q_2 = \ldots q$	q	$\frac{q}{t}$
$t_1 = t_2 = \ldots t$		

The estimation of the q_j parameters is somewhat more difficult. During the early stages of design or development one can try and estimate q_j using historical data. One way to derive the q_j parameter is to obtain failure rate data on the program and equate it to z_o using the assumptions of case 3 or 4 of Table 1; and solve for q_j. This process can be repeated as the program is written and better values for q_j determined. There is a possibility that one could calculate q_j from a more basic procedure. Knuth has shown that most FORTRAN statements are relatively simple and fall into one of several classes. If each of these classes also has a characteristic error rate, then by analysis of the q_j values for several examples, we should be able to derive characteristic values for the q_j parameters.

3.7.6 Conclusions

The models developed above allows one to decompose a program into a number of modules, paths, modes, or other functional entities. One can then compute an expression for the software failure rate in terms of probabilistic and deterministic parameters which can be estimated from historical data or determined by analysis or experiment. The model provides a clear cut procedure for relating the reliability of a large software system to the reliability of its constituent parts. The model is presently being applied to a number of modest size problems in order to obtain typical parameter values and validate the model.

References

1. M. Shooman, "Probabilistic Models for Software Reliability Prediction", published in Probabilistic Models for Software, Freiberger Editor, Academic Press, N.Y. 1972, p. 485-502.

2. J. Jelinski and P. B. Moranda, "Software Reliability Research," (Same source as Ref.1).

3. M. L. Shooman, "Structural Models for Software Reliability Prediction", Proceeding of the 2nd International Conference on Software Engineering, October 1976, San Francisco, California.

SECTION IV

DIRECTIONS FOR NEXT PERIOD'S WORK

In the next period we plan to work on the following:

1. Adamowicz: Further work on measures for the evaluation of software.

2. Baggi: Completion of a report on the construction of an automatic driver for Shooman's model of test covering each program path.

3. Laemmel: Continuation of studies on statistical program testing.

4. Marshall: Application of graph theory to statistical sampling approaches for software reliability.

5. Ruston and Berlinger: Applications of software physics to complexity measures.

6. Shooman and Popkin: Continuation of work on levels of program testing.

7. Shooman and Ruston: Experimental tests for (1) validation of seeding/tagging estimates (2) Shooman's extended debugging models (incorporating errors generated during the debugging process), and (3) obtaining data for verification of software physics and other complexity measures.

SECTION V

PROFESSIONAL ACTIVITIES

This section summarizes the professional activities of the research personnel working on this contract.

5.1 Published and Submitted Papers and Reports

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

2. S. N. Mohanty and M. Adamowicz, "Proposed Measures for the Evaluation of Software," to appear in Proceedings of the Symposium on Computer Software Engineering, 1976.

3. 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.

4. 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.

5. 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.

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

7. 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.

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

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

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

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

12. 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.

13. M. Horodniceanu, E. Cantilly, M. Shooman, L. Pignataro, "Transportation System Safety Methodology," First Year Final Report, November 1976, U.S. Dept. of Transportation, Contract DOT-05-50241.

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

15. B. Rudner, "Seeding/Tagging Estimates of the Number of Software Errors," Poly EE/EP Report, November 1976.

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

17. M. Shooman and S. Natarajan, "Effect of Manpower Deployment and Bug Generation on Software Error Model," Poly EE/EP 76-007, SMART 102, May 1976.

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

19. M. 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.

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

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

22. A. Laemmel and M. Shooman, "Statistical (Natural) Language Theory and Computer Program Complexity," Poly EE/EP Report, Spring 1977.

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

24. L. Shaw and M. Shooman, "Confidence Bounds and Propagation of Uncertainties in Systems Availability and Reliability Computations." Naval Res. Logistics Quarterly (to appear).

5.2 Talks and Seminars

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

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

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

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

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

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

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

8. M. L. Shooman and S. Natarajan, "Effect on Manpower Deployment and Error Generation on Software Reliability," MRI Symposium on Computer Software Engineering, 1976.

9. 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.

5.3 Symposia and Technical Societies

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

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

5.4 Honors and Awards

1. Professors C. Marshall, H. Ruston and M. Shooman, are listed in Who's Who in the East.

2. Professor M. Shooman is listed in Who's Who in America.

3. Professor M. 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.

5.5 Committees

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

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

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

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

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

☆U.S. GOVERNMENT PRINTING OFFICE: 1977-714-025/181

METRIC SYSTEM

BASE UNITS:

Quantity	Unit	SI Symbol	Formula
length	metre	m	...
mass	kilogram	kg	...
time	second	s	...
electric current	ampere	A	...
thermodynamic temperature	kelvin	K	...
amount of substance	mole	mol	...
luminous intensity	candela	cd	...

SUPPLEMENTARY UNITS:

plane angle	radian	rad	...
solid angle	steradian	sr	...

DERIVED UNITS:

Acceleration	metre per second squared	...	m/s
activity (of a radioactive source)	disintegration per second	...	(disintegration)/s
angular acceleration	radian per second squared	...	rad/s
angular velocity	radian per second	...	rad/s
area	square metre	...	m
density	kilogram per cubic metre	...	kg/m
electric capacitance	farad	F	A·s/V
electrical conductance	siemens	S	A/V
electric field strength	volt per metre	...	V/m
electric inductance	henry	H	V·s/A
electric potential difference	volt	V	W/A
electric resistance	ohm		V/A
electromotive force	volt	V	W/A
energy	joule	J	N·m
entropy	joule per kelvin	...	J/K
force	newton	N	kg·m/s
frequency	hertz	Hz	(cycle)/s
illuminance	lux	lx	lm/m
luminance	candela per square metre	...	cd/m
luminous flux	lumen	lm	cd·sr
magnetic field strength	ampere per metre	...	A/m
magnetic flux	weber	Wb	V·s
magnetic flux density	tesla	T	Wb/m
magnetomotive force	ampere	A	...
power	watt	W	J/s
pressure	pascal	Pa	N/m
quantity of electricity	coulomb	C	A·s
quantity of heat	joule	J	N·m
radiant intensity	watt per steradian	...	W/sr
specific heat	joule per kilogram-kelvin	...	J/kg·K
stress	pascal	Pa	N/m
thermal conductivity	watt per metre-kelvin	...	W/m·K
velocity	metre per second	...	m/s
viscosity, dynamic	pascal-second	...	Pa·s
viscosity, kinematic	square metre per second	...	m/s
voltage	volt	V	W/A
volume	cubic metre	...	m
wavenumber	reciprocal metre	...	(wave)/m
work	joule	J	N·m

SI PREFIXES:

Multiplication Factors	Prefix	SI Symbol
1 000 000 000 000 = 10^{12}	tera	T
1 000 000 000 = 10^{9}	giga	G
1 000 000 = 10^{6}	mega	M
1 000 = 10^{3}	kilo	k
100 = 10^{2}	hecto*	h
10 = 10^{1}	deka*	da
0.1 = 10^{-1}	deci*	d
0.01 = 10^{-2}	centi*	c
0.001 = 10^{-3}	milli	m
0.000 001 = 10^{-6}	micro	μ
0.000 000 001 = 10^{-9}	nano	n
0.000 000 000 001 = 10^{-12}	pico	p
0.000 000 000 000 001 = 10^{-15}	femto	f
0.000 000 000 000 000 001 = 10^{-18}	atto	a

* To be avoided where possible.

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.