# FOURTH ANNUAL PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

**November 10-11, 1986**

**Hilton Hotel**
**Portland, Oregon**

# TABLE OF CONTENTS

**Chairman's Message** . . . . . iv

**Conference Committee** . . . . . v

**Authors** . . . . . vi-vii

**Exhibitors** . . . . . viii

**Keynote (Abstract and Biography)** . . . . . ix
*"Limits on What We Can Do With Software"*
David L. Parnas, Queen's University

**Session 1. Evaluation Techniques** . . . . . 1

*"A Risk Driven Approach to Program Development Strategy"* . . . . . 3
Denis C. Meredith, Consultant

*"Unproductive Software Testing"* . . . . . 23
William Hetzel, Software Quality Engineering

*"Metrics of Software Verification"* . . . . . 31
Genard T. Catalano, Kemp Leetsch, George Aziz, Boeing Commercial Airplane Co.

**Session 2. Design Analysis** . . . . . 51

*"Automating Structured Analysis"*. . . . . 53
Andy Simmons, Alan Hecht, Cadre Technologies, Inc.

*"Team Systems Analysis: A Methodology for Validating Software Analysis and Design"* . . . . . 61
Albert F. Case, Jr., Nastec Corporation

*"Experience Using a Static Analyzer to Improve Software Quality in a Large System"*. . . . . 95
Lawrence A. McCarter, SAIC (Science Applications International Corporation)

**Session 3. Test Tools** . . . . . 115

*"UDT: A Tool for Debugging and Testing Software Units"*. . . . . 117
Bruno Alabiso, Motasim Najeeb, Craig Thomas, Intel Corporation

*"TCL and TCI: A Powerful Language and an Interpreter for Writing and Executing Black Box Tests"* . . . . . 147
Arun Jagota, Vijay Rao, Intel Corporation

*"T: The Automatic Test Case Data Generator"* . . . . . 167
Robert M. Poston, PEI (Programming Environments, Inc.)

# TABLE OF CONTENTS (continued)

**Session 4. Metrics** . . . . . 177

*"QA Metrics in the Workplace"* . . . . . 179
Dave Dickmann, Hewlett Packard

*"Software Quality Measurement - A Methodology and its Validation"* . . . . . 189
James L. Warthman, Computer Science Innovations, Inc.

*"A Comparison of Counting Methods for Software Science and Cyclomatic Complexity"* . . . . . 225
Nancy Currans, Hewlett Packard

**Session 5. Release/Version Control** . . . . . 249

*"Data Control in a Maintenance Environment"* . . . . . 251
Michael W. Evans, Expertware, Inc.

*"Proposal for a Software Design Control System (DCS)"* . . . . . 265
Robert Babb II, Richard Hamlet, Oregon Graduate Center

*"A Programmer's Database for Reusable Modules"* . . . . . 273
T. G. Lewis, Oregon State University, I. F. Eissa, Cairo University, Egypt

**Session 6, Panel: Integrated vs. Independent Software Evaluation** . . . . . 293

Chuck Asberry, Tektronix; Rich Martin, Intel Scientific Computers; David A. Rodgers, Boeing Commercial Airplane Co.; David Gelperin, Software Quality Engineering; and LeRoy Nollette, Panel Chairman

# Chairman's Message

MONIKA HUNSCHER

Welcome to the Fourth Annual Pacific Northwest Software Quality Conference. We are pleased that you took advantage of this opportunity to share knowledge and ideas on "Software Excellence."

In the computer industry, where time to market and product life are continually decreasing, "Software Excellence" is particularly significant, because only through the conscious effort of producing high-quality software products will we continue to satisfy the demand for our products as well as be able to solve problems as yet unspecified.

The Proceedings contains 15 papers from software engineers and managers who responded to our Call for Papers. This year's conference includes more exhibits by selected vendors who offer products which will aid us in achieving our goal. A new element of the conference is a series of workshops which provide in-depth knowledge related to our theme.

We welcome your comments on this year's conference and your suggestions for the 1987 program.

# Conference Committee

## CONFERENCE OFFICERS/COMMITTEE CHAIRS

Monika Hunscher - President and Chairman; *Floating Point Systems, Inc.*

Sue Strater - Technical Program; *Mentor Graphics Corporation*

Steve Shellans - Workshops; *Tektronix, Inc.*

Sue Bartlett - Secretary; *Engineering Systems Products*

Bill Edmark - Vice-President, Exhibits; *Intel Scientific Computers*

Richard A. Martin - Vice-President, Exhibits; *Intel Corporation*

Dale Mosby - Public Relations; *Sequent Computer Systems, Inc.*

LeRoy Nollette - Panel Chairman

Dennis Schnabel - Treasurer; *Mentor Graphics Corporation*

## CONFERENCE PLANNING COMMITTEE

Paul Blattner, *Quality Software Engineering*

Genard Catalano, *Boeing Computer Services*

Dave Dickmann, *Hewlett Packard*

S. Kea Grilley, *Intel Corporation*

G. W. Hicks, *Test System Strategies*

Arun Jagota, *Intel Corporation*

David Kerchner, *Floating Point Systems*

Ben Manny, *Intel Corporation*

Kenneth Oar, *Hewlett Packard*

Janet Sheperdigian, *Intel Corporation*

Joseph Soehardjiono, *Intel Corporation*

Wayne Staley, *Floating Point Systems*

Ron Swingen, *Mentor Graphics*

Craig Thomas, *Intel Corporation*

## PROFESSIONAL SUPPORT

Lawrence & Craig, Inc. - *Conference Management*

Liz Kingslien - *Graphic Design*

Classic Exposition - *Decorator for Exhibits*

# Authors

Bruno Alabiso
Intel
5200 N.E. Elam Young Parkway
Hillsboro OR 97124-6497

George Aziz
Boeing Commercial Airplane Co.
PO Box 3707, MS 05-52
Seattle WA 98124

Robert Babb II
Oregon Graduate Center
19600 N.W. Von Neumann Dr.
Beaverton OR 97006

Albert F. Case, Jr.
Nastec Corporation
24681 Northwestern Highway
Southfield MI 48075

Genard T. Catalano
Boeing Commercial Airplane Co.
PO Box 3707, MS 05-52
Seattle WA 98124

Nancy Currans
Hewlett Packard
3422 N.W. Maxine Circle
Corvallis OR 97330

Dave Dickmann
Hewlett Packard
1000 N.E. Circle Blvd.
Corvallis OR 97330

I. F. Eissa
Computer Science Department
Oregon State University
Corvallis OR 97331

Michael W. Evans
Expertware, Inc.
2685 Marine Way, #1209
Mountain View CA 94043-1125

David Gelperin
Software Quality Engineering
3015 Hartley Road, Suite 16
Jacksonville FL 32217

Richard Hamlet
Oregon Graduate Center
19600 N.W. Von Neumann Dr.
Beaverton OR 97006

Alan Hecht
Cadre Technologies
222 Richmond Street
Providence RI 02903

Bill Hetzel
Software Quality Engineering
3015 Hartley Road, Suite 16
Jacksonville FL 32217

Arun Jagota
Intel H-251
5200 N.E. Elam Young Parkway
Hillsboro OR 97124-6497

Kemp Leetsch
Boeing Commercial Airplane Co.
PO Box 3707, MS 05-52
Seattle WA 98124

T. G. Lewis
Computer Science Department
Oregon State University
Corvallis OR 97331

Lawrence A. McCarter
SAIC
1213 Jefferson Davis Hwy, #1500
Arlington VA 22202

Denis C. Meredith
Consultant
2042 Kathy Way
Torrance CA 90501

Motasim Najeeb
Intel
5200 N.E. Elam Young Parkway
Hillsboro OR 97124-6497

David L. Parnas
Computer Science
Queen's University
Kingston, Ontario, Canada

# Authors (continued)

Robert M. Poston
Programming Environments, Inc.
4043 State Highway 33
Tinton Falls NJ 07753

Andy Simmons
Cadre Technologies
222 Richmond Street
Providence RI 02903

Vijay Rao
Intel H-251
5200 N.E. Elam Young Parkway
Hillsboro OR 97124-6497

Craig Thomas
Intel
5200 N.E. Elam Young Parkway
Hillsboro OR 97124-6497

James L. Warthman
Computer Science Innovations
220 S.W. Clearmont
Palm Bay FL 32905

# Exhibitors

Cadre Technologies, Inc.
Contact: Robin Chapman
222 Richmond Street
Providence RI 02903
(401) 351-5950

Cipherlink Corporation
Contact: Larry Mull
3232 Nebraska
Santa Monica CA 90404
(213) 828-8331

Expertware, Inc.
Contact: Gary Furr
2685 Marine Way, Suite 1209
Mountain View CA 94043-1125
(415) 965-8921

IBST
Contact: Catherine Kirkham
536 Weddell Drive, Suite 7
Sunnyvale CA 94089
(408) 745-1237

Interactive Systems Corporation/
GEC Software
Contact: Patrick Moore
2401 Colorado Ave., 3rd Floor
Santa Monica CA 90266
213/453-8649

KnowledgeWare, Inc.
Contact: Diana Felde
2006 Hogback Road
Ann Arbor MI 48105
(313) 971-5363

Northwest Instrument Systems, Inc.
Contact: Ted Gary
PO Box 1309
Beaverton OR 97075
(503) 690-1300

Perennial
Contact: Barry Hedquist
4677 Old Ironsides Drive, Suite 450
Santa Clara CA 95054
(408) 727-2255

Promod, Inc.
Contact: Thomas L. Scott
22981 Alcalde Dr.
Laguna Hills CA 92653
714/855-8560

Software Quality Engineering
Contact: Sandra L. Wasser
3015 Hartley Road, Suite 16
Jacksonville FL 32217
(904) 268-8639

Specialized Systems Consultants
Contact: Belinda Frazier
PO Box 55549
Seattle WA 98155
(206) 367-8649

Symbolics, Inc.
Contact: Pete Gasparelli
135 Lake Street S., Suite 240
Kirkland WA 98033
(206) 822-4595

Tektronix, Inc.
Contact: Rodney Bell
PO Box 4600, MS 92-680
Beaverton OR 97075
(503) 629-1933

Teledyne Brown Engineering
Contact: Melanie Fisher
MS 202, Cummings Research Park
Huntsville AL 35807-5301
205/532-1661

# Keynote

## LIMITS ON WHAT WE CAN DO WITH SOFTWARE

**Dr. David L. Parnas**
**Professor Computer Science**
**Queen's University, Kingston, Ontario, Canada**

Large software systems - those consisting of millions of lines of code - present special problems, not only in the design and implementation, but also in the testing. One of Dr. Parnas' special interests is the exploration of the limits of our ability to build trustworthy software. His widely publicized resignation from President Reagan's SDI ("Star Wars") advisory panel was a response to the untestability of SDI software.

### Biography

Dr. David Lorge Parnas, born 10 February 1941 in Plattsburgh, New York, is Professor of Computer Science at Queen's University in Kingston, Ontario, Canada. Professor Parnas also leads the Software Cost Reduction Project at the Naval Research Laboratory in Washington, D.C. He has taught at Carnegie-Mellon University, the University of Maryland, the Technische Hochschule Darmstadt, the University of North Carolina at Chapel Hill, and the University of Victoria in British Columbia. Dr. Parnas is interested in all aspects of software engineering. His special interests include program semantics, language design, program organization, process structure, process synchronization, and precise abstract specifications. Because of his widely publicized involvement in the U.S. SDI project, he has been studying the limits on our ability to build reliable software. He is currently leading an experimental redesign of a hard-real-time system in order to evaluate a number of software engineering principles. He is also involved in the design of a language involving new control structures and abstract data types.

# Session 1

## EVALUATION TECHNIQUES

*"A Risk Driven Approach to Program Development Strategy"*
Denis C. Meredith, Consultant

*"Unproductive Software Testing"*
William Hetzel, Software Quality Engineering

*"Metrics of Software Verification"*
Genard T. Catalano, Kemp Leetsch, George Aziz, Boeing Commercial Airplane Co.

# A RISK DRIVEN APPROACH

# TO

# PROGRAM DEVELOPMENT STRATEGY

Denis C. Meredith
2042 Kathy Way
Torrance, California 90501

Abstract:

Most software development organizations fail to allocate sufficient resource to permit testing as thoroughly as theory says is appropriate. In view of this limitation, it would be advisable to devote more of the resources available to those areas where they can produce the maximum benefit. This paper suggests a way to identify high risk modules at design time, using this information to sequence development in such a way that modules requiring the most testing are available to be tested for the longest time.

Biography:

Denis Meredith is an independent consultant, concentrating in the areas of Software Testing and Quality Assurance; Tools Selection and Implementation; and Project Selection, Scheduling, and Management. He has often been asked to speak on these and other data processing topics for various groups such as the DPMA, AUUA, CIPS, and EDPAA chapters.

Mr. Meredith graduated from the United States Naval Academy at Annapolis with a B.S. degree, and has completed several graduate courses in Business Administration at Pennsylvania State University, where he also taught computer courses. He has been a member of the IEEE Unit Test Standards and Life Cycle Process Standards Committees, and holds CDP and CSP certificates. Denis has been published in Data Management and has written for Auerbach's Data Processing Management Series.

## I. Introduction

There is almost never enough resource available to do as much testing as theory says is appropriate. In view of this limitation, the question is how can the resource available be used in the most effective way; i.e., in such a way that those faults which have the most potential for damage and those which are most likely to occur are found?

As a general rule, modules which are written first are tested most. If it were possible to identify modules which needed the most testing, then a development strategy could be selected which required modules to be written in a sequence based on the relative amount of testing needed for each. A technique called risk analysis can be used to evaluate each module, once designed, to determine its development priority relative to other modules included in the design unit.

## II. Risk Analysis

The term risk has a variety of meanings depending on who uses it and in what context. For example, to auditors risk means exposure to loss or damage to an organization through misuse or improper function of a system, or through intrusion or unauthorized use of a system. Auditors usually express risk as potential annual dollar loss.

In the Data Processing environment, risk assessment is coming to be used at three levels: to assist in strategic planning, project management, and more recently to determine a development strategy.

### Strategic Planning

No organization has the resources available to develop all applications that may be useful or desirable. A risk assessment procedure can be used to quantify the relative risk of development projects under consideration. This quantification of risk can then be used in conjunction with other factors to determine which projects should be included in the portfolio of projects underway at a given time. The mix of projects appropriate for an organization will depend on such things as the experience and stability of the development staff, the criticality of the proposed projects to the future success of the organization, the users' perception of Management Information Systems (MIS) capabilities, and the "venturesomeness" of the organization.

Another use of the project risk score at this level is in planning the sequence of projects depending on the importance of the application in terms of payback and how quickly the organization wants to move into technologically risky areas.

## Project Management

To a development project manager, risk refers to the chance that a system being developed will be delivered late, go over the authorized budget, or will fail to meet user expectations.

The ability to establish the relative risk of all projects being considered can help to focus management attention where it is most likely to be needed. Several organizations are using risk assessment techniques to determine what level of conformance to development life-cycle standards will be required or the number and depth of project reviews that will be required.

An assessment of risk is also a useful datum to be presented to a steering committee as part of a decision package which would include estimates of effort, payback, and cost.

## Development Strategy

For the purpose of selecting a development strategy, risk will be defined as the potential that undetected faults remain in software because of inadequate testing. These potential faults could possibly cause:

- catastrophic system failure in production (abnormal termination or ABEND) resulting in emergency repair and rerun (or other contingency)
- failure to comply with government regulations
- economic loss through improper payment or billing and/or resulting legal action
- inappropriate business decisions made based on incorrect or improperly presented information
- loss of data

Two categories of risk have been identified, "Failure Impact" and "Fault Likelihood." Failure impact risk considers the cost or damage which could occur to the organization if a failure occurs. In general, it will rely on the users' evaluation of what's important in terms of frequency of use and value of results. Fault likelihood risk considers such things as complexity and new technology which contribute to the difficulty in implementing a design.

The following procedures were developed in response to the need of a particular client who recognized the problem of limited testing resource and asked the question "How can we be sure that we are putting our efforts in the right place?" This seemed like a reasonable question, and certainly it had been asked (and answered) before. A search of the available literature revealed some published work regarding the first two categories (strategic planning and project management), but only one unpublished reference to the use of risk assessment to prioritize testing efforts, and that was a fairly simplistic method intended for use on programs which were changed. The challenge, then, was to find a way to quantify the relative risk of a set of modules which were to be developed by a given development team. The method used had to be early, simple, and credible.

Early, in this case, meant at or before the completion of module design. This time-frame is necessary because any later consideration means that coding has already started - prioritizing the sequence of coding is then moot.

The method chosen had to be simple, or it would be perceived as too much of a chore by people who were concerned with "getting things done", not more paperwork.

The method had to be believable by experienced developers, including their supervisors, or the results could be rigged to reflect their biases.

The methodology developed followed the questionnaire technique that was already in use for risk assessment at the project level. The questions chosen were selected based on their real and perceived ability to predict risk, when in the development cycle they could be answered, and the ease with which they could be answered. (It was intended that most objections to the use of the procedures would be avoided before they were raised, rather than having to be overcome once the procedure was put in place.)

The set of seventeen questions was chosen from a much larger set of possible questions based on the criteria described above and some additional considerations such as:

- If the answer would be the same for all modules in the group, the question was discarded (project management and methodology questions)
- If the question was too complex or called for extra work or calculations, it was discarded (complexity metric for example)
- All the questions had to fit onto one piece of paper to avoid excuses for skipping the process

The assignment of weights to each question was somewhat arbitrary in that there was no database available on which to base the relative importance of each question. It was possible, however, to have the selected weights reviewed by a (small) number of experienced developers and a consensus was reached on the values after some discussion and adjustments.

The procedure described below is in use now on a large development project which is scheduled to be completed in 1988. Since there will be a phased implementation, some of the software will be in use as early as late 1986. Unfortunately, it will take several months of data collection before any meaningful patterns of software failure will appear. Even then, the results may be difficult to interpret. For example, suppose the programs identified as high risk using this technique show no significantly different failure patterns from those identified as low risk? That could be interpreted as a failure of the technique in that there was no difference between the two classes of programs. On the other hand, it could be interpreted as a success for the technique in that, because of increased attention paid to the high risk programs, the potential failures did not occur. Obviously, work in this area is far from complete.

## III. Procedures.

The attached questionnaire will assist in identifying high risk modules. Each question has a choice of several answers, each of which has a numerical value in the range of 0 - 3 associated with it. Some questions may be answered with more than one choice (they will be identified in the question explanations). Once the question is answered, the answer is multiplied by the weight and the result entered in the appropriate column, Impact or Likelihood. Some questions will, because of their nature, have entries in both columns. Again, these will be covered in the explanations.

Once design has progressed to the point that modules to be coded have been identified, the following procedure can be followed for each module:

Step 1. Identify general risks (if any) and document any particular concerns.

Step 2. Complete the risk questionnaire. You may comment on why any particular value was chosen.

Step 3. Calculate risk score. This will be the sum, for each of the two categories, of answer times weight for each question.

Step 4. When the questionnaire is complete for all modules, they can be ranked in order of risk score. Failure impact score should be considered first, then fault

likelihood score. The two scores cannot be combined in any meaningful way.

The key point to remember is that the objective of risk assessment at this level is to quickly evaluate the approximate relative risk of modules to be developed. This should not result in a numbers game which consumes scarce resources. There is nothing magic in the numbers generated - judgement must be exercised. Anything unusual may be noted on the questionnaire to document concerns not otherwise addressed.

## The Risk Questions Explained

1. Number of departments using

   How many independent groups of users will be accessing this module? (This indicates a diversity of requirements to be satisfied.)

| | |
|---|---|
| One | 1 |
| Two or three | 2 |
| More than three | 3 |

2. Sensitivity of data handled

   How vital are the results of the work done by this module to the user organizations?

| | |
|---|---|
| Useful | 1 |
| Important | 2 |
| Vital | 3 |

3. Kind of module

   What work does the module do? If more than one function is performed, add the values for all the functions before determining the score. Note that the answer to this question as well as some of the following ones are scored on both failure impact and fault likelihood.

| | |
|---|---|
| Inquiry | 1 |
| Report writer | 1 |
| Data extract/transfer | 2 |
| Algorithmic data manipulation | 2 |
| File/database update | 3 |
| Edit | 3 |
| Conversion | 3 |

4. Required response time

   How specific and tight are response time requirements? Data center standard is 5-10 seconds.

| | |
|---|---|
| Batch | 0 |
| No specific requirement | 1 |
| Data center standard | 2 |
| Specific requirement | 3 |

5. Number of transactions/day (batch or on-line)

Compared to other modules, how many transactions per day are expected to be processed by this module?

| | |
|---|---|
| Few (lower third) | 1 |
| Average (middle third) | 2 |
| Many (upper third) | 3 |

6. Degree of requirements or design changes

Compared to other modules, how many changes have been made or requested?

| | |
|---|---|
| None | 0 |
| Few (lower third) | 1 |
| Average (middle third) | 2 |
| Many (upper third) | 3 |

7. Required availability

How often will the program be run or what is its on-line availability requirement?

| | |
|---|---|
| Periodic use - weekly or less | 1 |
| Daily use but not 24 hrs/day | 2 |
| Required for 24 hrs/day use | 3 |

8. User experience with function in the past

How prone to error or difficult to use was the old way of doing the work, whether manual or automated?

| | |
|---|---|
| Routine | 1 |
| Average | 2 |
| Troublesome | 3 |

9. Possible error messages

Compared to other modules, how many possible different error messages can be produced by this module?

| | |
|---|---|
| None | 0 |
| Few (lower third) | 1 |
| Average (middle third) | 2 |
| Many (upper third) | 3 |

10. Number of modules linked or interfaced with

With how many other application modules does this module interface or accept input from/provide output to?

| | |
|---|---|
| Stand-alone | 1 |
| One or two | 2 |
| More than two | 3 |

11. Number of database segments or flat files accessed

How many logical views of the data does this module require?

| | |
|---|---|
| None | 0 |
| One | 1 |
| Two or three | 2 |
| More than three | 3 |

12. Language

In what language will the module be coded?

| | |
|---|---|
| Utility | 1 |
| ADS/on-line or batch | 1 |
| COBOL | 2 |
| Other | 3 |

13. Analyst experience with application

How much has the analyst worked in the past with this kind of application?

| | |
|---|---|
| Heavy (prior projects) | 1 |
| Light (understands concepts) | 2 |
| None (little or no exposure) | 3 |

14. Analyst experience with language

How much has the analyst worked with the language to be used on this module?

| | |
|---|---|
| Heavy (18 months or more) | 1 |
| Moderate (6 to 18 months) | 2 |
| Light (less than 6 months) | 3 |

15. Analyst experience with DBMS

How much has the analyst worked with the DBMS used by the module?

| | |
|---|---|
| DBMS not used | 0 |
| Heavy (18 months or more) | 1 |
| Moderate (6 to 18 months) | 2 |
| Light (less than 6 months) | 3 |

16. Size of module

Compared to other modules, how large is this module?

| | |
|---|---|
| Small (lower third) | 1 |
| Average (middle third) | 2 |
| Large (upper third) | 3 |

17. Complexity of logic

Compared to other modules, how complex is the logic required for this module?

| | |
|---|---|
| Simple (lower third) | 1 |
| Average (middle third) | 2 |
| Complex (upper third) | 3 |

## RISK QUESTIONNAIRE

Module name _____ Impact Wt Score Likelihood Wt Score

Module function _____

| | | Impact Wt | Impact Score | Likelihood Wt | Likelihood Score |
|---|---|---|---|---|---|
| **1. Number of departments using** | | | | | |
| One | 1 | | | | |
| Two or three | 2 | | | | |
| More than three | 3 | x 2 | ___ | | |
| **2. Sensitivity of data handled** | | | | | |
| Useful | 1 | | | | |
| Important | 2 | | | | |
| Vital | 3 | x 3 | ___ | | |
| **3. Kind of module** | | | | | |
| Inquiry | 1 | | | | |
| Report writer | 1 | | | | |
| Data extract/transfer | 2 | | | | |
| Algorithmic data manipulation | 2 | | | | |
| File/database update | 3 | | | | |
| Edit | 3 | | | | |
| Conversion | 3 | x 1 | ___ | x 4 | ___ |
| **4. Required response time** | | | | | |
| Batch | 0 | | | | |
| No specific requirement | 1 | | | | |
| Data center standard | 2 | | | | |
| Specific requirement | 3 | x 2 | ___ | x 3 | ___ |
| **5. Number of transactions/day (batch or on-line)** | | | | | |
| Few (lower third) | 1 | | | | |
| Average (middle third) | 2 | | | | |
| Many (upper third) | 3 | x 3 | ___ | x 1 | ___ |
| **6. Degree of requirements or design changes** | | | | | |
| None | 0 | | | | |
| Few (lower third) | 1 | | | | |
| Average (middle third) | 2 | | | | |
| Many (upper third) | 3 | x 3 | ___ | x 5 | ___ |
| **7. Required availability** | | | | | |
| Periodic use - weekly or less | 1 | | | | |
| Daily use but not 24 hrs/day | 2 | | | | |
| Required for 24 hrs/day use | 3 | x 4 | ___ | x 3 | ___ |
| **8. User experience with function in the past** | | | | | |
| Routine | 1 | | | | |
| Average | 2 | | | | |
| Troublesome | 3 | | | x 4 | ___ |

| | | Impact Wt Score | Likelihood Wt Score |
|---|---|---|---|
| 9. Possible error messages | | | |
| None | 0 | | |
| Few (lower third) | 1 | | |
| Average (middle third) | 2 | | |
| Many (upper third) | 3 | x 2 | ___ |
| 10. Number of modules linked or interfaced with | | | |
| Stand-alone | 1 | | |
| One or two | 2 | | |
| More than two | 3 | x 3 | ___ |
| 11. Number of database segments or flat files accessed | | | |
| None | 0 | | |
| One | 1 | | |
| Two or three | 2 | | |
| More than three | 3 | x 3 | ___ |
| 12. Language | | | |
| Utility | 1 | | |
| ADS/on-line or batch | 1 | | |
| COBOL | 2 | | |
| Other | 3 | x 1 | ___ |
| 13. Analyst experience with application | | | |
| Heavy (prior projects) | 1 | | |
| Light (understands concepts) | 2 | | |
| None (little or no exposure) | 3 | x 2 | ___ |
| 14. Analyst experience with language | | | |
| Heavy (18 months or more) | 1 | | |
| Moderate (6 to 18 months) | 2 | | |
| Light (less than 6 months) | 3 | x 3 | ___ |
| 15. Analyst experience with DBMS | | | |
| DBMS not used | 0 | | |
| Heavy (18 months or more) | 1 | | |
| Moderate (6 to 18 months) | 2 | | |
| Light (less than 6 months) | 3 | x 2 | ___ |
| 16. Size of module | | | |
| Small (lower third) | 1 | | |
| Average (middle third) | 2 | | |
| Large (upper third) | 3 | x 3 | ___ |
| 17. Complexity of logic | | | |
| Simple (lower third) | 1 | | |
| Average (middle third) | 2 | | |
| Complex (upper third) | 3 | x 4 | ___ |
| | | TOTALS | |
| COMMENTS: _____ | | Impact | Likelihood |
| _____ | | _____ | _____ |

A RISK DRIVEN APPROACH

TO

PROGRAM DEVELOPMENT STRATEGY

Denis C. Meredith

---

## RISK ANALYSIS

PROBLEM:

Not enough resources to test all software equally and completely.

## RISK IMPLIES:

- A future event
- Uncertain outcome
- Potential for loss

---

## RISK: Exposure to consequences

In a D P project, measure of likelihood of:

- Failure to obtain expected benefits
- Excessive implementation cost
- Excessive implementation time
- Inadequate technical performance
- Hardware/Software incompatibility
- Inadequate financial controls

RISK ASSESSMENT is the process of identifying and describing risk, evaluating both the probability of occurrence and potential impact, and providing a plan of action and resources to monitor occurrence and control consequences.

---

PURPOSE:

examine

probe

assess

NOT

offer solutions

develop strategies

Make risk visible so it can be managed

## WHERE DOES RISK ASSESSMENT FIT IN?

Short-Term, Low-Level

- Development, integration, and testing strategy

Medium-Term, Medium-Level

- Project management strategy
- Project evaluation and selection
- Project portfolio management

Long-Term, High-Level

- Long range planning
- Optimizing project mix

---

## OPTIMUM PROJECT MIX

Limited resources prevent development of all possible applications

Procedure to optimize mix:
- Rate proposed projects on risk
- Rate proposed projects on payback
- Determine organizational posture
- Choose mix

All high-risk projects jeopardize success of MIS and existance of organization

All low-risk projects minimize ROI

"Proper" mix depends on venturesomeness of the organization

* Feasibility Studies - Senior managers and MIS managers can agree that risk is in line with corporate goals
  - Are the benefits great enough to offset the risk?
  - Can the affected parts of the organization survive if the project fails?
  - Have alternatives been considered?

* Determine frequency and depth of project reviews

---

## RISK ANALYSIS

PROBLEM:

Not enough resources to test all software equally and completely.

QUESTION:

How to prioritize modules that need the most attention?

PROPOSED:

Risk Questionnaire - ask about easily determined early indicators.

STRATEGY:

Identify high-risk modules early so risky modules can be coded first, tested first, integrated first.

## RISK MANAGEMENT

- Technical Risk (Fault Likelihood)

  Where things are most likely

  to go wrong.

- Operational Risk (Failure Impact)

  What are the things that are

  most damaging if they happen.

---

BEWARE:

The difficulty with any quantitative risk assessment technique is that it is easy to lapse into a numbers game where the production of figures gives the appearance of accuracy even though the figures have little or no basis in fact.

## Bibliography

Burril, Claude W., and Ellsworth, Leon W., "Modern Project Management", B.E. Associates, 1980

Gelperin, David, and Hetzel, William, "Systematic Software Testing", DPMA Education Foundation, 1985

"Managing the Applications Development Project", IBM Corporation, 1977

McFarlan, F.Warren, "Portfolio approach to Information systems", Harvard Business Review, September-October 1981, pp.142-150

Perry, William L., "A Standard for Testing Application Software", Auerbach Publishers, Inc., 1985

Sill, Nancie L., "QA Development Reviews", System Development, June 1982

"The Dallas Tire Case", HBS Case Services, 1980, No. 9-180-006

# UNPRODUCTIVE SOFTWARE TESTING

by
Bill Hetzel
Software Quality Engineering
March 1986

## Abstract

While major improvements in techniques and in understanding and standardizing the software testing process have been achieved during the last few years, little or no progress is evident toward developing metrics to measure testing effectiveness and productivity. This paper explores the question of test effectiveness with particular emphasis on identifying productive and unproductive testing practices. A set of seven ineffective test practices are identified and described. Examples of productivity metrics are discussed and suggestions for more productive testing practices and metrics are offered.

## Biographical Sketch

BILL HETZEL is a principal in SOFTWARE QUALITY ENGINEERING based in Jacksonville, Florida. Software Quality Engineering is a consulting firm with a special emphasis on better software engineering and quality assurance. Services offered include training and consulting support in software validation, verification and testing methods as well as effective management and control of the software testing and quality assurance functions.

Bill is well-known as a software testing consultant and regularly presents industry seminars on testing and test management. He has helped many companies to develop and implement better testing standards and procedures and has been involved in numerous projects reviewing test plans and designs and consulting on testing problems and test effectiveness. He has written several recent books including **The Complete Guide to Software Testing**, published by Q.E.D. Information Sciences in 1984 and **Computer Information Systems Development: Management Principles & Case Studies**, published by South Western Publishing Company in 1985. In addition, Bill has written many articles and papers on software testing and engineering practices and is a frequent speaker at various conferences and professional meetings. Bill is a graduate of Brown University and earned his M.S. degree from Rensselear Polytechnic Institute and the Ph.D. in Computer Science from the University of North Carolina.

Contact at: SOFTWARE QUALITY ENGINEERING
3015 Hartley Road, Suite #16
Jacksonville, Florida 32217
(904) 268-8639

# UNPRODUCTIVE SOFTWARE TESTING

by
Bill Hetzel
Software Quality Engineering

## INTRODUCTION

What is a "productive" software test? Would you as a Manager be able to distinguish a high productivity effort from a low productivity one?

Testing in most projects is poorly understood and is not subject to effectiveness or productivity measurements. The most common practice is for testing to be judged indirectly. Project managers allocate resources to the test efforts and hope that the testing gets completed as planned. When the project goes well, testing is implicitly viewed favorably; when the software fails or when cost and schedule run over budget, then testing is viewed unfavorably.

Such indirect judgments regarding testing effectiveness are often inaccurate. While testing effectiveness correlates reasonably well with delivered product quality, poor quality does not imply poor or unproductive testing and good testing does not imply or assure good delivered quality. More direct measurement of testing's role within a project is necessary and should be an important objective for today's professional software manager.

## THE TESTING CONTRIBUTION

The testing activity contribution begins as soon as requirements are laid down and "tests" for those requirements begin to be developed. Requirements based tests provide "models" for how the eventual system will function and contribute to the project verification and validation effort. Early preparation and design of these tests helps everyone to understand the requirements more clearly and eliminates front-end misunderstandings. Examples always help us to understand better and test cases are wonderful "examples" of system behavior (given this set of inputs, here is what you can expect as outputs). If developed early they are powerful communication and validation tools and contribute to the project by isolating early requirements faults.

As a project moves into system design, the test activity continues to contribute tangible benefits. Test work during design should help to maintain a clear focus on requirement understanding, traceability, and acceptance. Fuzzy and unclear requirements are brought out into the open as efforts to design tests for them proceed. Effective

test strategy also surfaces risk areas in the designs and defines tests and checks that are to be built into the software permanently or that need to be tested directly before further detailed design and engineering effort is pursued.

As code is developed and tests can be executed, testing begins to take on its more recognized contribution of fault finding. Defects and faults are detected as the tests are run. As the testing proceeds through the various levels (unit, integration, system and acceptance) the software is undergoing constant change and correction. Productive testing seeks to reuse test cases and to save time and checking effort through well-designed and highly automated test procedures. With effective planning the test set becomes a "product" of the testing effort that is a deliverable for use over the entire software life cycle each time the software is modified. Testing may be thought of as producing software just as the project does. The software produced are automated test sets or reusable measurement instruments that are designed in such a way to effectively address system coverage and give us a meaningful measurement result whenever they are applied.

To summarize, effective testing provides several major contributions to a project:

- -- verification of requirements and design work (if cases are prepared early enough)
- -- finding software faults
- -- repeatable measurement of software quality and risk

These benefits are achieved through the design and construction of measurement instruments (test sets) that are a basic product of the test process.

## PRODUCTIVE AND UNPRODUCTIVE TESTING

During the past few years, we have asked many managers attending our classes to share how they look at testing in their projects and to describe particularly productive or unproductive practices they had experienced.

The following is a list of several particularly ineffective testing practices. While our goal is productive testing, it is helpful to focus on some of the poor practices as a way of identifying what we should not be doing. The list helps us to understand better how testing productivity might be measured and provides insight into the factors influencing testing effectiveness.

## Seven Ineffective Testing Practices

***Throw-Away Tests*** - The prevailing practice is to prepare tests, execute them and then throw them away when the program seems to be working. When the program is later modified, new tests are constructed and once again not saved for reuse. This low productivity practice stems from failing to recognize that testing produces a product. Test sets that have been "designed" from the start as reusable "products" with the goal of effective software measurement permit testing to be executed over and over again during the life cycle and pay for themselves many times over. Test set designs as well as the test sets may be used repeatedly and more effective as well as more productive testing is the result.

***Poor Coordination Across Test Levels*** - A second frequently observed practice is the development of tests for each test level independently with little coordination or opportunity to reuse tests across the levels. Unit tests are used only in unit testing; systems tests only in system testing; and acceptance tests only in acceptance testing. Such practices are inefficient and provide little additional risk management.

The productive strategy consists of starting first by defining the acceptance test set, then using it as a base to develop the system test set and then taking from that as much as possible to support unit or component testing. This coordinated back-end first design strategy ensures maximum reuse of test cases across levels and is much more productive than the every level designed on its own approach.

***Manual Tests*** - An especially low productivity practice is when tests require cumbersome manual checking or review to determine if actual results meet expectations. In one organization, over half the total time devoted to testing was being spent looking at output and writing and running special one-time utility or data base extract routines to check the test results. With good use of comparator tools and proper test set design, this percentage should run well under ten percent and approaches zero percent with a commitment to fully self-checking tests.

Testing on-line interactive transactions affords a good example of how using the right tool makes a difference. The normal practice is to prepare "test scripts" that spell out sequences of specific interactions with the application being tested. To conduct the test, the tester sits at the terminal and follows the script step by step, entering the keystrokes or function keys specified and checking the screen response for particular actions and results. Testing with such scripts is extremely labor-intensive and has the added burden of uncertainty

as to what was actually entered. Special tools that allow captured on-line test inputs as well as the expected screen responses and

then allow the script to be executed automatically are the solution to this low-productivity practice. Such tools give us the capability to design and implement automated test sets for on-line applications that may be used over and over again without any manual test effort.

Tests that have been designed so that they may be executed automatically allow for repeated use and can save great amounts of time and effort in the long run. While designing automated tests requires considerably more initial test preparation effort, the savings is returned in reduced execution support effort and the elimination of manual-labor-intensive and error-prone checking of test results.

***Testing Poor Software*** - A curious form of low productivity testing behavior is seen when software is tested prematurely and most of the tests fail. Nothing is gained by running test sets against unstable and severely faulty software other than wasted effort and energy. Failure to plan for and control the test start criteria properly leads to this problem. Once encountered the solution is not throwing more test resources into the fray but rather to back off and ensure the software has met the entrance criteria before beginning the testing again.

***Testing After Coding*** - The timing of <u>when</u> tests are designed during the project is a critical productivity variable. Tests developed before the software has been designed provide important requirements verification and validation support and are much more productive than tests prepared after construction. Rather than "code and test", we should be saying "test and code". Getting the tests first has a major influence on the resulting total productivity.

***Testing Without Calibration*** - The design and construction of calibrated test sets that cover the requirements systematically is difficult and time-consuming. However, the alternative of testing with a quickly-put-together "mass" of test cases that have never been correlated to the software requirements is no shortcut. This is a little like trying to take someone's temperature with a thermometer that has no markings on it. You may learn a little, but in the long run the measurement is unreliable and the result is low productivity--not quick results.

A special form of uncalibrated test set that may appear to be calibrated arises when the test effort is organized using code coverage as a completion criteria. There is no research to show that such a criteria is productive or useful. While it is measurable (and I applaud measuring it), we feel caution should be taken to prevent the criteria from becoming a palliative. There is no substitute for the hard work of test design to assure requirements coverage. Code coverage is nice, but not at the expense of requirements coverage!

***Extended Acceptance Tests*** - The acceptance test in most organizations is the responsibility of the user organization. The user may be told that he has full responsibility for system acceptance and should not sign off until his organization is ready for implementation and has thoroughly tested the system performance. That responsibility assignment is fine--however, if it is not coupled with a parallel requirement for planning and completing the test expeditiously, the results may be protracted and drawn out acceptance test periods that extend for months. The high productivity approach is to drive toward short demonstration-oriented acceptance tests that are planned and carried out under appropriate schedule and budget constraints.

## MEASURING TESTING PRODUCTIVITY

Having considered some of the test practices that contribute to high and low productivity, we are ready to explore how we might want to define and evaluate testing productivity. What is a productive test and what can we use to help managers and practitioners measure it?

A number of productivity measures have been tried. Some examples are the following:

***Ratio of Faults Found to Effort Expended*** - Testing is viewed as oriented toward discovering faults or errors in the software. The more faults found for a given expenditure of test effort, the more productive the test.

Faults are easy to find when the software is full of them and hard otherwise. Counting faults as a measure is naive at best, especially since one of the contributions testing is supposed to offer is to prevent them from arising and propagating.

***Ratio of Tests Completed to Effort Expended*** - The job of testing is to run tests. A simple count of the number run becomes the measure of productivity. The more tests run or prepared per unit of test time, the more productive the testing.

This measure ignores complexity differences between different tests and counting tests designed or completed would seem to reward work for work's sake. It is not the number of tests that matters, but whether they are the "right" or "best" set of tests for the resources and risk available.

***Comparison of Activity Against The Plan*** - A baseline is established through a test plan or project plan document. Productivity is then measured relative to the baseline. Test work completed ahead of time or under budget is productive, behind or over budget is unproductive.

Measuring the testing work against a plan or budget ignores the question of whether the plan was well thought out and is completely independent of the end-product software quality.

***Estimate of Faults Remaining*** - The objective of testing is to produce quality software. Testing productivity is measured indirectly by evaluating the delivered software fault density in comparison to the test effort consumed or expended. If a lot of test effort is expended, but many faults remain, then productivity is low. If effort is reasonable and few faults remain, then productivity is high.

The problem with faults remaining is that the true number becomes known only long after software delivery, and a low or high number remaining may bear little relationship to the real productivity or effectiveness of the test effort applied.

All of these measures suffer from fairly obvious and significant limitations. We feel that the major problem in developing decent measures has been the lack of an established process model for how the testing job is carried out. In our Software Quality Engineering testing courses, we teach a methodology for testing that breaks up the total activity into three major phases--test planning, acquiring or building the test set; and measurement or execution of tests.

This basic testing methodology should be used as the framework for measuring productivity. What I am suggesting is that it does not make sense to try to measure overall testing productivity, but instead, we should look at each phase separately. In the planning phase, the measure should relate the cost and time of getting a plan completed to the complexity and criticality of the project and test effort. In the acquisition phase, productivity should be defined by the effort and cost to build the test set, and in the measurement phase, it should be defined by tests that are run and completed.

We are currently experimenting with this suggested approach to defining and measuring testing productivity. We believe that efforts to measure and make testing productivity more visible and controllable are very important and worthwhile. More work and research is needed and we challenge others to pursue this area and help achieve the visibility and better measurement required for significant future improvement in testing.

# METRICS OF SOFTWARE VERIFICATION


## Abstract

Test Coverage analysis is an important area of work in software testing. While the primary analysis task is a black box one, that is to show that all requirements are tested, it is still necessary to carry out White Box coverage analysis on vendor-supplied software. White Box coverage addresses the question of how completely the code is exercised. Several terms are currently employed to describe the level of White Box coverage. Software testing tools and techniques used to check Boeing 737-300 Autopilot software will be discussed, including Inspection and Review, Design Reviews, Code Reviews and Test Plan Reviews. Unit (module) testing will be discussed which involves White Box (Structural) Testing. System (Functional) testing includes Black Box Functional testing, Coverage Analysis and Gray Box testing. Several testing tools will be discussed including invasive (instrumentation) and non-invasive tools. A set of coverage tools developed by Boeing and Sperry to analyze the Autopilot software will be described.

Genard Catalano received the Bachelor of ElectricaL Engineering degree from the City College of New York and Master of Science of Electrical Engineering from the University of Rhode Island and Ph.D. degree from Arizona State University. He worked three years for the Underwater Systems Center in New London, Connecticut as an Electrical Engineer, a Development Engineer for Motorola in Phoenix, Arizona, from 1968 to 1970. He was an Assistant Professor of Engineering from 1973 to 1977 at San Francisco State University where his research was in Biomedical Instrumentation. In 1977 he joined the staff at Seattle University as an Assistant Professor of Electrical Engineering. Since 1979 he has been with the Boeing Company. For the past three years he has been working in the area of Software Tools, concentrating on Automatic Methods of verifying instruction coverage.

**Present Mailing Address:**

Genard T. Catalano, Ph.D.
Boeing Commercial Airplane Company
P.O. Box 3707
Seattle, WA 98124
(206) 237-1891
(206) 783-4350

Kemp Leetsoh reoeived a BSEET degree from DeVRY Institute of Teohnology in 1986. He is ourrently employed by the Boeing Commeroial Airplane Company in the Software Team designing the Autopilot for the 737-300 airplane.

**Present Mailing Address**

Kemp Leetsoh
Boeing Commercial Airplane Company
P.O. Box 3707 MS:70-29
Seattle, WA. 98124
(206) 237-1891

George Aziz received the MSEE degree from the University of Washington in 1977. For the past 12 years he has participated in design of Avionics equipment for the Boeing Company. He is currently a member of the Software Team developing Autopilot Flight Director Systems for the 737-300 Airplane.

**Present Mailing Address:**

George Aziz
Boeing Commercial Airplane Company
P.O. Box 3707
Seattle, WA 98124
(206) 237-1891

# METRICS OF SOFTWARE VERIFICATION

Genard T. Catalano
Kemp Leetsch
George Aziz
Drew Shore*

Boeing Commercial Airplane Company
P. O. Box 3707 MS 70-29
Seattle, WA 98124

*Sperry Corporation
Flight Systems
P. O. Box 2111
Phoenix, AZ 85036

## 1. INTRODUCTION

Test Coverage analysis is an important area of work in software testing. While the primary analysis task is a "black box" one, that is to show that all requirements are tested, it is still important to carry out "white box" coverage analysis. White box coverage addresses the question of how completely the code is exercised. This paper describes the testing terminology used and some automatic software tools that were developed to test the Boeing Autopilot software.

Unit (Modules) Testing will be discussed which involves White Box (Structural) Testing. System (Functional) Testing including Black Box Functional Testing, Coverage Analysis and Gray Box Testing. Several dynamic testing tools will be discussed including invasive (instrumentation) and non-invasive tools. A set of coverage tools for instructional and path coverage developed by Boeing and Sperry respectively, to analyze the Autopilot software will be described.

## 2. BLACK BOX MODULE TESTING

This type of testing ensures that the results from executing the software module are the results expected using a set of input test values. The expected results could be achieved by use of the Software Design Specification (i.e., Design Implementation Document).

In Black Box Testing one ascertains that the outputs are correct with the knowledge of what the inputs are. The internal module implementation structure and the means at which the output was computed is of no consequence. The correctness of the results is reflected in a comparison of the expected results with the actual results.

On this Autopilot Verification Review Project, Black Box Testing was performed by means of a checklist as follows: (1) Is the module under test clearly defined with the proper version number? (2) Are the CPDD (i.e., computer program definition document which is the software design specification) references included and accurate? (3) Is the test specification properly identified? (4) Are the test inputs and expected results completely defined? (5) Is the test sequence or procedures included in the test report? (6) Are all stubs defined? (7) Did all the tests pass? These questions were answered by manual inspection of the test results and the software documents mentioned in (2) However (1), (4) and (7) were performed automatically.

## 3. WHITE BOX MODULE TESTING

In White Box Testing, an analysis on the software is performed based on the structure of the code. In other words, the code is analyzed "as implemented" for internal completeness (e.g., coverage, how completely is the code exercised?) and characterize the program behavior. Table 1 indicates a comparison between Black and White Box Testing. Applications of White Box Testing are Instruction Coverage, Branch and Path Coverage.

### A. INSTRUCTION (STATEMENT), VARIABLE AND CONSTANT COVERAGE

This type of coverage is a form of logic coverage. Logic coverage is a measure of how well the internal control flow logic of a program has been exercised. Instruction coverage is defined as a determination of whether or not all instructions of the code were executed. In Variable Coverage, one ensures that all the input and output variables passed to the module are used and exercised (i.e., change value at least once) during execution. One also ensures that all constants are used.

### B. BRANCH (DECISION) COVERAGE

This type of coverage ensures that all branches of a conditional instruction are executed. Alternately, each code segment between decision points is executed.

### C. PATH COVERAGE

This type of coverage ensures that every path through the code is executed. The definition of a "path" is not universally established however. A path may be defined as a set of instructions from one decision point to another or as a complete set of executable instructions from entry to exit [1].

TABLE I COMPARISON OF BLACK BOX AND WHITE BOX ANALYSIS

Black Box Analysis

* Inputs and processes construct the outputs with no regard to internal module implementation structure.

* All inputs shall be exercised at least once in each equivalent class.

* Both paths of equivalent DID "If-Then-Else" requirements shall be exercised at least once.

* Data at each input shall be set to each limit (i.e., +max/-min and zero) at least once.

* Discontinuities (i.e., boundaries) of the input or output shall be exercised at just less than, equal to and just greater than the value at the discontinuity.

* In the case of Boolean logic, show that a logic reversal has been shown to have occurred when (1) each input of each DID "and" gate shall be toggled "true-false-true" while all other inputs to that gate are "true" and (2) each input of each DID "or" gate shall be toggled "false-true-false" while all other inputs to that gate are "false" and (3) likewise for the inputs to "nand" gates and "nor" gates, only reversed logic of (1) an (2) respectively and (4) inputs to "XOR" gates shall be exhaustively toggled, if reasonable (3 inputs).

White Box Analysis

* Examine structure of module in terms of path and data structures.

* Show that the structure reflects the DID requirement:

  - I/O data
  - Data Structure and Formats
  - Module Instructions
    - sequence
    - branching
    - data usage
    - data transformation
      - arithmetic
      - logic
  - Initialization of Module
  - Interrupt Support
  - Data/Data Interface
  - BCAC Software Standard

* Show that the MMI testing at a minimum:

  - executes all instructions
  - uses all the data in the data structures

* Test all decision instructions with conditions:

  (1) less than
  (2) equal to
  (3) greater than the tested condition

## 4. APPLICATIONS USING AUTOMATIC TOOLS

### A. BLACK BOX TESTING

A comparison of files containing expected and actual results was performed using a program written in Pascal. The first of two columns indicate the variable name and the expected value respectively for the test cases used. The latter two columns indicate the actual variable name and the numerical results, respectively. The fifth and sixth columns are the error detection columns, which detect a variable name or numerical value error respectively.

Another form of Black Box Testing was to write a HOL (Higher Order Language) program using the Design and Implementation Drawing (DID which is the document used to write the software) and compare the output variable results with the module test results. Figure 1 shows the sample DID page of the Chapin Diagram that was used to write the Pascal Program for Module MMTRCOL. The test inputs were provided by the module test data. The output of the HOL program is shown in Figure 2A for three of the output variables. Figure 2B indicates the output from executing the actual module code. Note that the last three columns of Figure 2A and 2B are identical. Figure 2 also indicates the paths, in the second column, taken through the Chapin Diagram of Figure 1 (paths not shown in the Figure).

### B. WHITE BOX TESTING

#### 1. INSTRUCTION, BRANCH AND VARIABLE COVERAGE - APPLICATION I, COVERAGE OF CPU1 AUTOPILOT COMPUTER

In the Autopilot there are two microprocessors. Two programs were written for each language The microprocessors are designated as CPU1 and CPU2. Both programs are run on the Boeing VAX computer and partially written in DCL (Digital Command Language). Figure 3 shows a partial trace file that is generated in the simulator. This file is searched and forms an output table of addresses along with the instructions executed in ascending order. A counter then checks how many times the instruction was executed and displays a table that shows a count pattern which must remain consistent to verify all instructions were executed in the correct order. Figure 4 shows the output coverage table. The branch instructions show jump locations and times of jump execution to verify all path

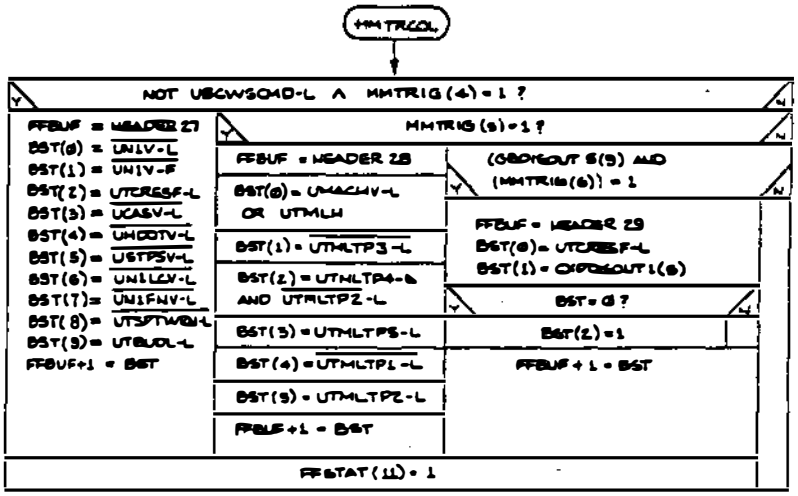



Figure 1. Chapin Diagram used to generate hol program.

| FFSTAT | FFBUF | FFBUF+1 | TEST# | PATHS | FFSTAT | FFBUF | FFBUF+1 |
|---|---|---|---|---|---|---|---|
| 0000 | 0000 | 0000 | 1 | EE | 0000 | 0000 | 0000 |
| 0800 | 41EC | 0001 | 2 | AA | 0800 | 41EC | 0001 |
| 0800 | 41EC | 0002 | 3 | AA | 0800 | 41EC | 0002 |
| 0800 | 41EC | 0004 | 4 | AA | 0800 | 41EC | 0004 |
| 0800 | 41EC | 0008 | 5 | AA | 0800 | 41EC | 0008 |
| 0800 | 41EC | 0010 | 6 | AA | 0800 | 41EC | 0010 |
| 0800 | 41EC | 0020 | 7 | AA | 0800 | 41EC | 0020 |
| 0800 | 41EC | 0040 | 8 | AA | 0800 | 41EC | 0040 |
| 0800 | 41EC | 0080 | 9 | AA | 0800 | 41EC | 0080 |
| 0800 | 41EC | 0100 | 10 | AA | 0800 | 41EC | 0100 |
| 0800 | 41EC | 0200 | 11 | AA | 0800 | 41EC | 0200 |
| 0800 | 41F0 | 0001 | 12 | BB | 0800 | 41F0 | 0001 |
| 0800 | 41F0 | 0021 | 13 | BB | 0800 | 41F0 | 0021 |
| 0800 | 41F0 | 0022 | 14 | BB | 0800 | 41F0 | 0022 |
| 0800 | 41F0 | 0004 | 15 | BB | 0800 | 41F0 | 0004 |
| 0800 | 41F0 | 0028 | 16 | BB | 0800 | 41F0 | 0028 |
| 0800 | 41F0 | 0010 | 17 | BB | 0800 | 41F0 | 0010 |
| 0800 | 41F0 | 0020 | 18 | BB | 0800 | 41F0 | 0020 |
| 0800 | 41F4 | 0004 | 19 | CC | 0800 | 41F4 | 0004 |
| 0800 | 41F4 | 0001 | 20 | DD | 0800 | 41F4 | 0001 |
| 0800 | 41F4 | 0002 | 21 | DD | 0800 | 41F4 | 0002 |
| 0800 | 41F4 | 0003 | 22 | DD | 0800 | 41F4 | 0003 |
| F7FF | AAAA | BBBB | 23 | EE | F7FF | AAAA | BBBB |
| 0000 | CCCC | DDDD | 24 | EE | 0000 | CCCC | DDDD |
| 0000 | EEEE | FFFF | 25 | EE | 0000 | EEEE | FFFF |
| | (A) | | | | | (B) | |

FIGURE 2. COMPARISON OF OUTPUT RESULTS OF A HOL PROGRAM SHOWING THE (A) EXPECTED RESULTS FROM THE PROGRAM AND (B) THE ACTUAL RESULTS FOR MODULE MMTRCOL.

```
**************************************************
*                TESTCASE 7                      *
**************************************************
   LABEL          EXPECTED          ACTUAL          PASS/FAIL

   A      X      Y      Z      B      C      D      E     PAGE  ABSOLUTE ADDRESS
 5DBE   0000   1016   4009   4CEC   CCCC   1555   0000    000         400C
   J      K      L      W      S      M      T      P      INSTRUCTION
 0000   0000   8000   0000   0030   0000   7000   400C    LDR   A,4000

   A      X      Y      Z      B      C      D      E     PAGE  ABSOLUTE ADDRESS
 0E39   0000   1016   4009   4CEC   CCCC   1555   0000    000         400E
   J      K      L      W      S      M      T      P      INSTRUCTION
 0000   0000   8000   0000   0030   0000   7000   400E    LDR   B,4001

   A      X      Y      Z      B      C      D      E     PAGE  ABSOLUTE ADDRESS
 0E39   0000   1016   4009   06D4   CCCC   1555   0000    000         4010
   J      K      L      W      S      M      T      P      INSTRUCTION
 0000   0000   8000   0000   0030   0000   7000   4010    LDR   C,1000

   A      X      Y      Z      B      C      D      E     PAGE  ABSOLUTE ADDRESS
 0E39   0000   1016   4009   06D4   CCCC   1555   0000    000         4012
   J      K      L      W      S      M      T      P      INSTRUCTION
 0000   0000   8000   0000   0070   0000   7000   4012    JSIPP $-8

   A      X      Y      Z      B      C      D      E     PAGE  ABSOLUTE ADDRESS
 0E39   0000   1016   4009   06D4   CCCC   1555   0000    000         3000
   J      K      L      W      S      M      T      P      INSTRUCTION
 0000   0000   8000   0000   0070   0000   7002   3000    RTRNP

** BREAKPOINT ADDRESS ENCOUNTERED ** EXECUTION INTERRUPT **
   A      X      Y      Z      B      C      D      E     PAGE  ABSOLUTE ADDRESS
 0E39   0000   1016   4009   06D4   CCCC   1555   0000    000         3000
   J      K      L      W      S      M      T      P      INSTRUCTION
 0000   0000   8000   0000   0070   0000   7002   3000    RTRNP

   A      X      Y      Z      B      C      D      E     PAGE  ABSOLUTE ADDRESS
 0E39   0000   1016   4009   06D4   CCCC   1555   0000    000         4013
   J      K      L      W      S      M      T      P      INSTRUCTION
 0000   0000   8000   0000   0070   0000   7000   4013    RTR   B,A

   A      X      Y      Z      B      C      D      E     PAGE  ABSOLUTE ADDRESS
 0E39   0000   1016   4009   0E39   CCCC   1555   0000    000         4014
   J      K      L      W      S      M      T      P      INSTRUCTION
 0000   0000   8000   0000   0030   0000   7000   4014    NEG   A,A

   A      X      Y      Z      B      C      D      E     PAGE  ABSOLUTE ADDRESS
 F1C7   0000   1016   4009   0E39   CCCC   1555   0000    000         4015
   J      K      L      W      S      M      T      P      INSTRUCTION
 0000   0000   8000   0000   0070   0000   7000   4015    STRD  1016

   A      X      Y      Z      B      C      D      E     PAGE  ABSOLUTE ADDRESS
 F1C7   0000   1016   4009   0E39   CCCC   1555   0000    000         4017
   J      K      L      W      S      M      T      P      INSTRUCTION
 0000   0000   8000   0000   0070   0000   7000   4017    LDR   A,1006

   A      X      Y      Z      B      C      D      E     PAGE  ABSOLUTE ADDRESS
 0000   0000   1016   4009   0E39   CCCC   1555   0000    000         4019
   J      K      L      W      S      M      T      P      INSTRUCTION
 0000   0000   8000   0000   00B0   0000   7000   4019    CMR   A,1008
```

Figure 3. Sample of a trace file generated in the CPU1 Simulator during a module test.

conditions were taken correctly by following the pattern throughout the test.

The first section of the output file is instruction coverage data. The second section shows the use of constants by checking the computer listing of the module, and extracting the list of defined constants. This list of constants is then checked with the trace file to make sure all have been used during testing, or there were no constants in the module, a message appears at the end of the section for quick reference.

Section 3 of Figure 4 is a symbol table that utilizes the cross reference section of the module listing to verify all symbols were accessed. If the symbol was previously checked as a constant, no check is executed, and the test continues with the next valid symbol. The symbols are checked by scanning the trace file for the addresses assigned to them in the test driver map.

The symbol label, memory addresses, and times of execution display the symbols usage. If any of the symbols are not used, the number of errors is displayed at the bottom of the section. A summary of the data, which is a quick reference for path coverage, is displayed following the coverage data tables.

These are the conditions which would be considered as an untested module if all numbers don't display a zero in the coverage summary statements located at the end of the Tables in Figure 4. The programs must be updated as new versions of the Autopilot software are delivered. This is performed by editing one subprogram.

2. INSTRUCTION, BRANCH AND VARIABLE COVERAGE - COVERAGE OF CPU2 AUTOPILOT COMPUTER

A program was developed [2, 3] to check instruction, branch, variable and constant coverage of the Boeing 737-300 Autopilot software modules contained in the CPU2 computer. The process consists of running the test driver with the module under test and generating a trace file from the simulator, a post processor which analyzes the trace file in terms of the four factors mentioned above. The post processor generates a Summary Report which tests

```
CHECKING CPU1 MODULE * PVSGMD *

*****************************************
* 1.       INSTRUCTION TABLE            *
*****************************************
ABSOLUTE TIMES     TYPE    BRANCH    TIMES
ADDRESS  EXECUTED  BRANCH  LOCATION  BRANCHED
********:*********:******:********:*********
000C         7                          0
000E         7                          0
0010         7                          0
0012         7        BU     000A       7
0013         6                          0
0014         6                          0
0015         6                          0
0017         6                          0
0019         6                          0
001B         6        B      0029       3
001C         3                          0
001E         3                          0
0020         3        BU     000A       3
0021         2                          0
0023         2                          0
0024         2                          0
0026         2                          0
0027         2                          0
0029         5                          0
002B         5                          0
002C         5        B      0030       1
002D         4                          0
002F         4        BU     0035       4
0030         1                          0
0032         1                          0
0034         1        BU     000B       1
0035         5                          0
0037         5        BU     000A       5
0038         4                          0
003A         4                          0
003B         4                          0
003D         4                          0
003E         4                          0
0040         4                          0
0042         4        BU     000B       4
0043         2                          0
0044         2                          0
0046         2                          0
0048         2                          0
004A         2                          0
004C         2                          0
004E         2        B      0052       2
0052         2                          0
0054         2                          0
0055         2                          0
0057         2                          0

*********************************
* 2.     CONSTANT TABLE         *
*********************************
ABSOLUTE TIMES     CONSTANT
ADDRESS  EXECUTED  LABEL
********:*********:************

0000         1        PVUNIT
0001         1        SPM300
0002         1        SPM301
0003         1        SPK301
0004         1        SPK303

0005         1        SPK302
0006         1        SPK305
0007         1        SPK308
000A         3        SUBU1
000B         2        SUBU2
NUMBER OF ERRORS = 0
*********************************
* 3.     SYMBOL TABLE           *
*********************************
ABSOLUTE TIMES     SYMBOL
ADDRESS  EXECTUTED LABEL
********:*********:************
0005         1        GBFMDISWD
0008         1        FMCVSLIMI
000C         2        *PVSGMD
0012         2        GBATDISWD
0029         3        SPG2
0030         3        FMCTGTVS
0035         3        LOADB
0052         3        NORETARD
0055         2        GAMMA
1000         1        SPVTAS
1002         1        VERTC
1004         1        SHDOTC
1006         1        SPMODE
1008         1        SCMODE
100A         2        LEXEC10
100C         1        GSFMTVSCM
100E         1        SPGAMD
1010         1        SVSINP
1012         1        SAQINW
1014         1        SAQINT
1016         2        SPVRLM
1018         1        SPVSUS
101A         1        SPTP
3000         0        VGAIN
3001         0        LIMIT
NUMBER OF ERRORS = 2
*******************************
* 4.    COVERAGE SUMMARY      *
*******************************
1 CONDITIONAL BRANCHES ALWAYS TAKE
0 CONDITIONAL BRANCHES NEVER TAKEN
0 CONSTANTS NOT USED
2 SYMBOLS NOT USED
```

Figure 4. Output Coverage Table for CPU1 computer.

and documents the following items: (1) number of times every instruction is executed and flags an error if an instruction is not executed (2) number of times a branch is taken and indicates an error if a branch is not taken (3) lists all input and output variables and their values, in hex, during execution of the module, and flags the variable if it is not used, (4) the number of times the constants are used and flags an error. The end of the report summarizes the problem and the number of occurrences. A Summary is shown in figure 5. This report is of an artificial module and errors were intentionally inserted to reveal the type of history record that is produced by the post processor.

This coverage tool indicated the extent of integrity of the module testing. The results of running this tool was to identify such items as unreachable code, and constants variables not used during module test. The results of this test would indicate areas of testing that need modification in order to include these problem areas.

## 3. PATH ANALYSIS - APPLICATION I

An example of code instrumentation was performed on the Boeing 757/767 Autopilot software. The language used was called AED for "Automated Engineering Design". It was originally developed at Massachusetts Institute of Technology under U.S. Air Force sponsorship. It is a block structured language and is very similar to Pascal. Paths in the code were checked by inserting logic probes directly into the code string by means of additional instructions. An example of path probe insertion into the code is show in Figure 6. The path diagram is shown in Figure 7. A sample of the output is shown in Figure 8 indicating the paths checked. There are twelve test cases shown and sixteen paths checked. A "one" entry in the table indicates that particular path was tested. For example, path 1 was taken during test case numbers 1, 2, 3 and 4. The test cases chosen were the same ones chosen when the module was verified. This was done in order to check the extent of the coverage obtained by verification test case selections. When the probes are inserted, one must ascertain that the program flow was not modified. This was performed in two ways. One way is to have a

```
*NAME OF MODULE:  XYZ
                                                  # OF      # OF
                                               EXECUTIONS  JUMPS  BRANCH TO:
*DATE OF TEST RUN:
*10-OCT-1984
*14:03:37.28
*PROGRAM COUNTER
*-------------------------------------------------------------------------------
 1040                        XYZ                3          0
 1044  B                                        3          2  1054
 1046                                           1          0
 104A                                           1          0
 104E                                           1          0
 1052                                           1          0
 1054                        AP_GA              3          0
 1058                                           3          0
 105C                                           3          0
 1060                                           3          0
 1062                                           3          0
 1064                        TEMP2              3          0
 1068                                           3          0
 106A                                           3          0
 106E  B                                        3          3  1084      PROBLEM 1
 1070                                           0          0            PROBLEM 3
 1072  B                                        0          0            PROBLEM 3
 1074                                           0          0            PROBLEM 3
 107A  BU                                       0          0            PROBLEM 3
 107C                                           0          0            PROBLEM 3
 1082  BU                                       0          0            PROBLEM 3
 1084                                           3          0
 1088                                           3          0
 108A                                           3          0
 108C                        TEMP3              3          0
 108E                                           3          0
 1092                                           3          0
 1094                                           3          0
 1098  B                                        3          3  10AE      PROBLEM 1
 109A                                           0          0            PROBLEM 3
 109C  B                                        0          0            PROBLEM 3
 109E                                           0          0            PROBLEM 3
 10A4  BU                                       0          0            PROBLEM 3
 10A6                                           0          0            PROBLEM 3
 10AC  BU                                       0          0            PROBLEM 3
 10AE                                           3          0
 1082                                           3          0
 1084                        TEMP4              3          0
 1088                                           3          0
 108A                                           3          0
 108E                                           3          0
 10C0                                           3          0
 10C4  B                                        3          0  10CE      PROBLEM 2
 10C6                                           3          0
 10CA  B                                        3          0  10CE      PROBLEM 2
 10CC  BU                                       3          3  10DE
 10CE                        IC_INT             0          0            PROBLEM 3
 1000                                           0          0            PROBLEM 3
 10D4                                           0          0            PROBLEM 3
 10C8                                           0          0            PROBLEM 3
 10DC  BU                                       0          0            PROBLEM 3
 10CE                        INTEG              3          0
 10E2                                           3          0
 10E6                                           3          0
 108A                        BYPASS             2          0
 10EC                                           2          0
 10EE                                           2          0
 10F2                        QQ016              2          0
 10F6                                           2          0
 10F8  B                     QC014              8          6  110A
 10FA  B                                        2          1  1104
 10FC                                           1          0
 1102  BU                                       1          1  110A
 1104                        OVER1              1          0
 110A                        NEXT1              8          0
*       CONSTANT DATA
*******************************************
*******************************************
 110C                        QQ009              2
 1110                        QQ004              3
 111C                        QQ006              3
 1120                        QQ007              3
 1124                        QQ008              3
 1128                        LAG1               0                    PROBLEM 4

*                  TOTAL NUMBER OF ERRORS=  22

*OUTPUTS
*NAME OF  CONTENTS OF VARIABLE
*VARIABLE

QC001   : 00000000

QC002   : 00020002

QQ003   : 23C3FFEB

QC0048  : 28F3FFFF

QC0098  : VARIABLE NOT USED

QC011   : VARIABLE NOT USED

QC012   : BAAABAAA

QQ013   : VARIABLE NOT USED

TOTAL NUMBER OF OUTPUT
VARIABLES NOT USED=        3
*INPUT VARIABLES
INPUTS

EREQU   : 0000FFFF0001

QQ001   : VARIABLE NOT USED


QC002   : VARIABLE NOT USED


QC003   : VARIABLE NOT USED


QC0048  : VARIABLE NOT USED


QC0098  : 6516FFFF0001


QCD11   : 0000FFFF0001


QC012   : VARIABLE NOT USED


QC013   :     38E4     0001     FFFF


TOTAL NUMBER OF INPUT
VARIABLES USED= 5
* KEY
*B = CONDITIONAL BRANCH
*BU= UNCONDITIONAL BRANCH
*NUMBER OF PROBLEM#1= 2 CONDITIONAL BRANCH ALWAYS TAKEN
*NUMBER OF PROBLEM#2= 2 CONDITIONAL BRANCH NEVER TAKEN
*NUMBER OF PROBLEM#3=17 INSTRUCTION(S) NEVER EXECUTED
*NUMBER OF PROBLEM#4=1  CONSTANTS NOT USED
```

Figure 5. Output Summary Report indicating instruction coverage for the CPU2 Computer.

```
106:    COMMENT - ************************** $,
107:              FOR I = 0 STEP 1 UNTIL 11
108:              DO   BEGIN
109:                   PITCH100 = IN1(I);
110:                   GINTC = IN2(I);
111:                   PATH.PAC = IN3(I);
112:                   TPATH = IN4(I);
113:                   VERS.ROLL = IN5(I);
114:                   CPL.VERSINE = IN6(I);
115:                   PAMO = IN7(I);
116:                   TSYNC = IN8(I);
117:                   PAMAX = IN9(I);
118:                   DTHETE.L = IN10(I);
119:                   DTHETE.R = IN11(I);
120:                   AP.CMD.R = IN12(I);
121:                   AP.CMD.L = IN13(I);
122:                   IN.AIR = IN14(I);
123:                   IF TSYNC
124:                   THEN BEGIN
125:                        PATH1(I) = TRUE;
126:                        IF NOT AP.CMD.L
127:                        THEN BEGIN
128:                             PATH3(I) = TRUE;
129:                             IF AP.CMD.R
130:                             THEN BEGIN
131:                                  PATH5(I) = TRUE;
132:                                  T = DTHETE.R+DTHETE.R;
133:                                  END
134:                             ELSE BEGIN
135:                                  PATH6(I) = TRUE;
136:                                  T = 0.0;
137:                                  END;
138:                             END
139:                        ELSE BEGIN
140:                             PATH4(I) = TRUE;
141:                             T = DTHETE.L;
142:                             IF AP.CMD.R
143:                             THEN BEGIN
144:                                  PATH15(I) = TRUE;
145:                                  T = T+DTHETE.R;
146:                                  END
147:                             ELSE BEGIN
148:                                  PATH16(I) = TRUE;
149:                                  T = T+DTHETE.L;
150:                                  END;
151:                             END;
152:                        PTCH.CMD.R = PITCH100-T# 5;
153:                        GOUT( C, /T = /,T, C, /PTCH CMD.R =/,
154:                         PTCH CMD.R) $,
155:                        END
```

Figure 6. An example of instrumenting the AED language for path coverage. The PATH arrays are inserted in the code to indicate the conditional paths taken. Partial listing.

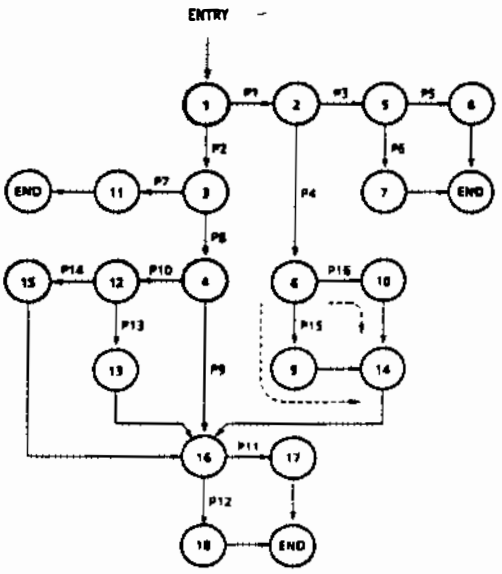



Figure 7. Path Flow Diagram for Module PITCHCMD for Test #3. The nodes indicate junctions and decisions in the code of the module during execution of the 12 test cases. All possible paths from entry to exit are also shown.

of the code structure so that you are certain that no modifications were made to the code structure. Another way the instrumentation was checked was to compare the output results with that of the non-instrumented code. This was accomplished by comparing the test output results of the instrumented and non-instrumented code.

For Boolean logic modules, it was necessary for the output of all gates to go to the high state at least once during the test cases. Figure 9 indicates a typical module that was instrumented for this purpose. The gate outputs were designated as nodes and numbered accordingly as shown in Figure 9. The test is considered successful if all nodes go high at least once during the test and that the resultant outputs are identical with the verification data. The results of the test runs are shown in Figure 10. It indicates all seven nodes go high at least once during the fourteen test cases.

## 4. PATH ANALYSIS - APPLICATION II

This path coverage analysis program breaks a module down into blocks. A block is defined as a contiguous set of instructions that end with an instruction that can change the program counter, (e.g., conditional or an unconditional jump). If a block falls through to the next block, it is considered one path. If the block jumps to a different block, it is considered a second path. The path coverage program analyzes each block to determine whether it falls through or branches, and totals the number of paths found. The program generates a number of reports including a Test Coverage Summary Report which tallies the module name, the number of test cases run, total paths found and percent coverage.

The percent coverage is defined as the paths executed divided by the total paths found. Another report lists the Path Execution Details which tabulate the paths tested and the number of times the path is executed during a specific test case.

| Test Case Number | | | | | | | | | | | | P# |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 7 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 8 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 9 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 10 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 11 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 12 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 13 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 14 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 15 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 16 |

Figure 8. Output path analysis file. Columns indicate test case numbers and the rows indicate path numbers.

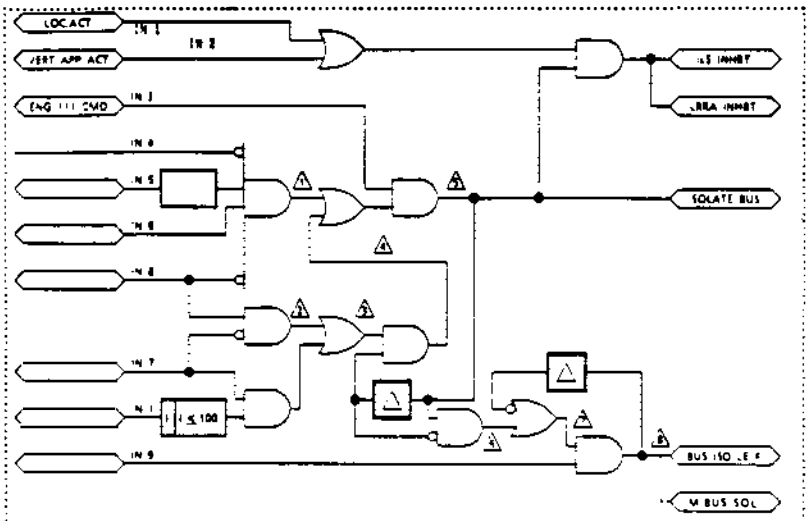



FIGURE 9. EXAMPLE OF LOGIC GATE COVERAGE

THE FOLLOWING ARE OUTPUT PROBE LEVELS

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 5 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 6 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 7 |

Figure 10. Output file indicating logic gate coverage. Columns are the number of test cases. The rows show the number of nodes checked.

It also yields the cumulative tally of paths over all test cases, and the percent Cumulative Coverage which is defined as the number of cumulative paths executed divided by the total number of paths found. Each test case exercises a portion of the total number of paths. The cumulative paths taken is the sum of th unique paths taken by each test case. It was proven that this coverage was comprehensive in that it included the number of independent Paths computed using the McCabe approach, the cyclomatic number V(G) [1].

## 5. CONCLUSION

A number of methods were discussed dealing with ways of testing software. Two broad categories are the Black and White Box test methods. Black Box testing entails more of an inspection method of assessing software quality by inspection of the test results and checking for (1) consistency and use of variables and constant names with the Software Specification Document (2) verify the actual results against the expected results (3) verifying consistency of the Software Control Document (major functional specification) with the Design Implementation Document (Software Design Specification).

In White Box testing, the code structure is inspected by means of path flow, statement or instruction coverage, and use of symbol variables and constants. A number of automatic methods were discussed, two of them indicate path flow and the total path coverage obtained during a complete module test. A second method consisted of determining the testing quality by instruction coverage. In addition, a branch analysis is performed by determining the number of times the branch is taken. Variables and constants are also documented and a summary of problems are indicated in the output record for both computers of the autopilot. The length of time these programs run depends on the size of the module and the number of test case. It varies from about one to thirty minutes on the VAX8600.

## 6. REFERENCES


(1) McCabe, T. J., "A Complexity Measure" IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, pp 308-320, December 1978.

(2) Catalano, G. T. and Rodgers, D. A., "Automatic Verification Analysis and Checking of Boeing 737 Autopilot Flight Software". Proceedings of Mini and Microcomputers in Control, Filtering, Signal Processing, Las Vegas, Nevada, December 10-12, 1984, pp 115-118.

(3) Catalano, G. T. and Aziz, G., "An Assessment of Software Quality Based on Testing for Coverage". Proceedings of Software and Hardware Applications of Microcomputers, February 5-7, 1986, pp. 185-189.

# Session 2

## DESIGN ANALYSIS

*"Automating Structured Analysis"*
Andy Simmons, Alan Hecht, Cadre Technologies, Inc.

*"Team Systems Analysis: A Methodology for Validating Software Analysis and Design"*
Albert F. Case, Jr., Nastec Corporation

*"Experience Using a Static Analyzer to Improve Software Quality in a Large System"*
Lawrence A. McCarter, SAIC (Science Applications International Corporation)

# Automating Structured Analysis

Andy Simmons and Alan Hecht
Cadre Technologies Inc.
222 Richmond St.
Providence, R.I. 02903
(401) 351-5950

## Abstract

Software developers have long been plagued by specification problems. Users have trouble communicating their requirements, and expensive errors can result from poor specifications. There is a lack of good estimation techniques and meaningful metrics, and it is difficult to adapt some software without starting from scratch. These problems compound as software projects get larger and more complex. Research in software engineering has shown the importance of requirements analysis and good specifications. However, many analysis techniques are manual and can be tedious. The additional effort spent at the beginning of a project can expedite the overall development process and improve the quality of the resulting system. Therefore, automating the analysis process should produce further productivity and quality improvements. We will discuss how some of the problems that confront software developers can be minimized. We will also attempt to show how automated analysis environments can benefit software developers, using the specific example of the **teamwork/SA®** system.

**Andy Simmons** has been with Cadre for one year as a Customer Support Representative. Prior to that he was with Lawrence Livermore National Laboratories for five years. He is a member of IEEE and ACM.

**Alan Hecht** has been with Cadre for three years, first as a Principal Software Engineer and currently as Technical Support Manager. Prior to that he was with Bell Telephone Laboratories for three years. He is a member of IEEE and ACM.

## The Software Development Process

Increased demand for complex systems has caused software development techniques to evolve into an engineering discipline. Controlling errors and managing production schedules has become more important. Some recent work has focused on gathering statistics from case studies of projects. At least half of the projects had problems that originated in the requirements or functional specification (see Figure 1). To help put this in perspective, we can view the software development process as divided into five phases: analysis, design, implementation, test and verification, and maintenance.

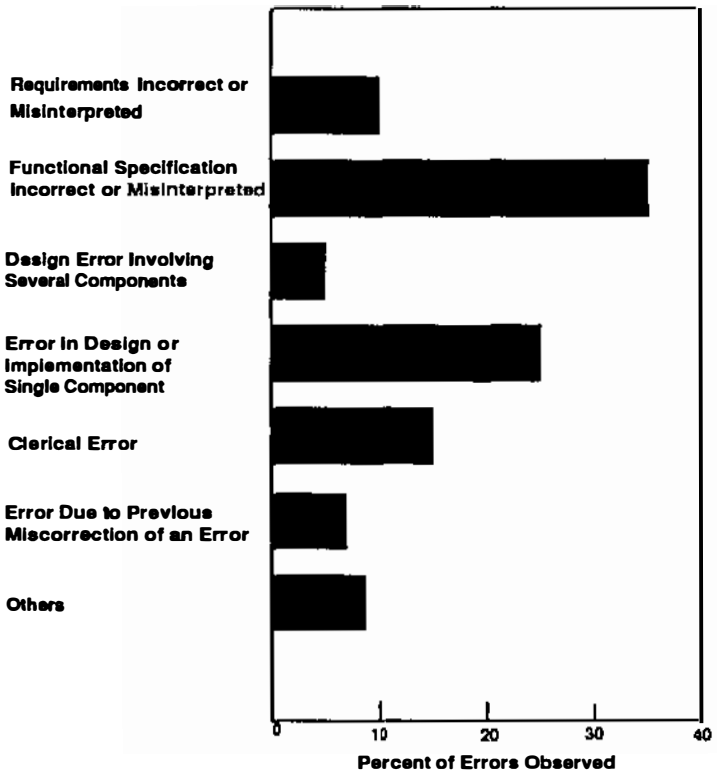



**Figure 1:** Sources of Errors[1].

The analysis phase is concerned with understanding *what* a system is supposed to do. It is supposed to be an *implementation independent* description or abstract view of the system to be developed. The product of analysis is a requirements specification (sometimes called a functional specification) that describes the system function and important constraints.

The design phase addresses *how* the system is to be implemented. It is concerned with the physical aspects of the system. The optimal structure of the various software modules and how they interface is determined. Ideally, the design information should be complete enough to reduce the implementation effort to little more than a translation to a target programming language.

The implementation phase is concerned with producing executable code. Knowledge of both the design and the

[1]Adapted from [Ramamoorthy 84]

target environment is incorporated to produce the final system software. All the physical aspects of the system are addressed during implementation.

Information from the previous three phases is used in the testing and verification phase. Test plans can be derived from specifications and designs [Boehm 84]. The testing phase verifies that the software conforms to the specification and that the code is correct. The best that test and verification techniques can do is prove that a program is consistent with its specification. They cannot prove that a program meets the user's desires [Wulf 80]. This means that extra care must be taken during analysis to insure that the specification is as complete as possible and is a correct reflection of what the user really wants. This can be accomplished through methods that support checks for consistency and clearly communicate system requirements. **Teamwork/SA** is one such method, and it will be discussed later in this paper.

Bug fixes and adaptations which result from experience with the software are activities of the maintenance phase. At this point the software is being used -- the ultimate test. Users will come across errors or suggestions as they gain experience with the software. Maintenance procedures must handle the orderly evolution of the code. They must insure that changes will not have deleterious effects on the system.

A study by [Boehm 84] showed that errors detected later in the development life cycle cost more to fix than errors detected during analysis (See Figure 2).

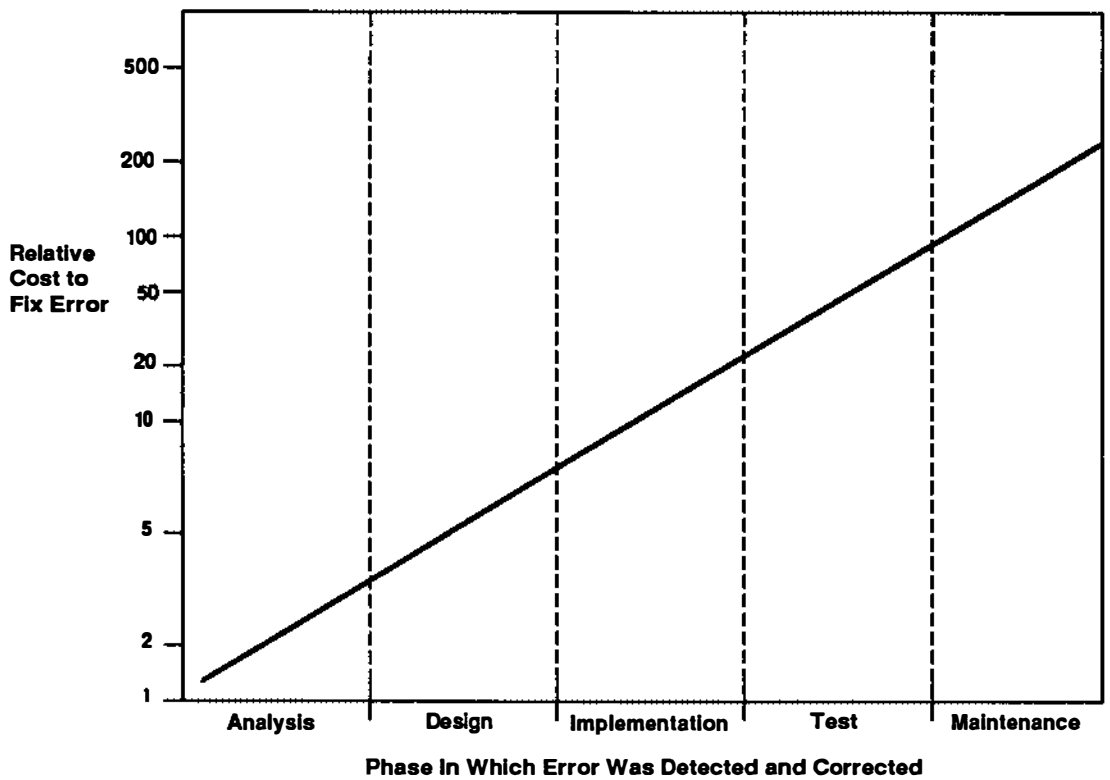



**Figure 2:** Cost of Error Versus When it is Detected[2].

[2]Adapted from [Boehm 84]

Figure 1, discussed previously, showed that the majority of errors in a software project can be traced to requirements and specification problems. These facts illustrate the value of spending more time at the beginning of a project, performing analysis. This can be difficult for programmers and users to accept as both may be anxious to see code being produced [Ramamoorthy 84]. These ideas have only recently become well understood and brought into practice.

Many approaches and methodologies utilize the concept of the software life cycle. In particular, structured analysis (which refers to several methods [Gane 79, DeMarco 78, Ross 77]) addresses the beginning phase of requirements analysis.

## Structured Analysis

Structured analysis views a system from the perspective of the data flowing through it. The function of the system is described by processes that transform the data flows. Structured analysis takes advantage of information hiding through successive decomposition (or top down) analysis. This allows attention to be focused on pertinent details and avoids confusion from looking at irrelevant details. As the level of detail increases, the breadth of information is reduced. The result of structured analysis is a set of related graphical diagrams, process descriptions, and data definitions. They describe the transformations that need to take place and the data required to meet a system's functional requirements.

De Marco's approach [DeMarco 78] consists of the following objects: *data flow diagrams, process specifications,* and a *data dictionary* (See Figure 3).

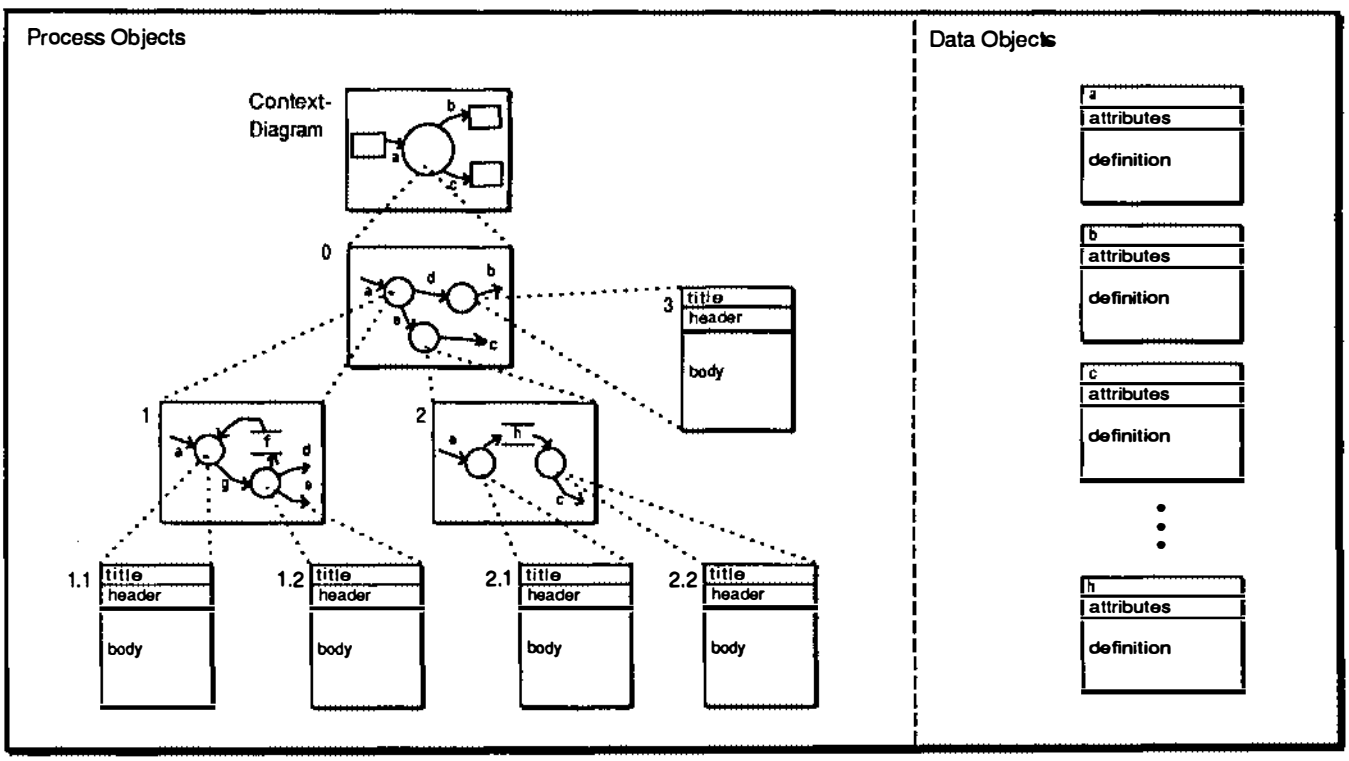



**Figure 3:** Model Objects

Data flow diagrams (DFDs) are directed graphs. The arcs represent data, and the nodes (circles or bubbles) represent processes that transform the data. A process can be further decomposed to a more detailed DFD which shows the subprocesses and data flows within it. The subprocesses can in turn be decomposed further with another DFD until their functions can be easily understood. Functional primitives are processes which do not need to be decomposed further. Functional primitives are described by a process specification (or mini-spec). The process specification can consist of pseudo code, flowcharts, or structured English. The DFDs model the structure of the system as a network of interconnected processes composed of functional primitives.

The data dictionary is a set of entries (definitions) of data flows, data elements, files, and data bases. The data dictionary entries are partitioned in a top-down manner. They can be referenced in other data dictionary entries and in data flow diagrams.

An analyst manipulates these objects to obtain a description of the functional requirements. Each object or diagram offers a particular view of some aspect of a system. It is often helpful to have several views side-by-side. This can help relay additional, contextual information that any single view does not possess. Once a model has been created, it is usually refined. The analyst must check consistency between the data flow diagrams and the definitions in the data dictionary. Successive iterations of the model result from communication between users and analysts. The resulting description contains contributions from the user and analyst, and is understood by both.

Many text and graphical diagrams may be required to describe large or complex systems. Graphics and text editors have been around for a long time. However, they do nothing to help insure that complex models are consistent and complete. Manual checking of syntax and consistency between various diagrams and definitions has proven to be quite tedious, especially on relatively large and complex projects. This expenditure of analyst time is often perceived to (and in some cases actually does) counteract any gains from performing analysis.

## Automating Structured Analysis

Hardware CAD/CAM systems have contributed to the development of systems with higher levels of complexity, performance and reliability, at costs previously unattainable through purely manual design efforts. This is sparking interest in automating the software development process. Generally, the value of automating structured analysis is not well understood. Automating these methods requires a large degree of symbolic manipulation, sorting, and searching operations. These operations can be computationally intensive. Computer graphics systems have only recently become practical as their cost to performance ratios have come down. Automated structured analysis tools require high performance computers and graphics capabilities. In the past, these systems were not considered cost effective for software development activities.

Studies show that faster response time is directly related to increases in productivity [Brady 85]. Trends in hardware are towards higher performance, smaller packages, and lower costs [Burger 84]. The demand for software, and for software developers, is increasing relative to their availability, making these people more valuable. The costs of systems required to effectively automate software development are shrinking. To realize a pay-back on the cost of automated analysis tools in a relatively short time (less than one year), only moderate individual productivity gains are necessary.

The economies involved in providing engineers with automated tools must be re-evaluated. The *workstation*, a desktop high performance microcomputer with high resolution graphics capabilities, is becoming increasingly more popular. These systems can fulfill the hardware requirements for automating structured analysis. Costs of individual workstation systems are coming down, making them competitive with larger, time-shared computer systems. Currently, many workstations support local area networks. This feature enables members of project teams to share information and communicate ideas efficiently. In configurations where many developers share a single host, there

is the risk of stopping all work if the host fails. If a single workstation in a network fails or if the network itself fails, it is possible for some work to continue. The perceived cost of workstations per seat may be relatively high (e.g., comparing a desktop workstation to a terminal). The actual net cost may be less than most conventional approaches.

## Team*work*/SA

Team*work*/SA is an automated tool for systems analysis. It can support many simultaneous users working on the same project or even many projects, and it takes advantage of features provided by the latest workstation technology. It offers complete support of the DeMarco structured analysis techniques. The DeMarco approach was chosen because it is the most widely known, taught, and used non-proprietary structured analysis method. Data flow diagrams are created using a syntax-directed editor that incorporates model building rules. Its interactive graphics package supports a high resolution bit-mapped display, mouse, and keyboard. Modern user interface techniques are used, including a multi-window display and context specific pop-up and pull-down menus.

Multiple, simultaneous views of a specification can be displayed by team*work*/SA (See Figure 4). It has simple commands for traversing through the various parts of a model. Model objects may be entered in any order.

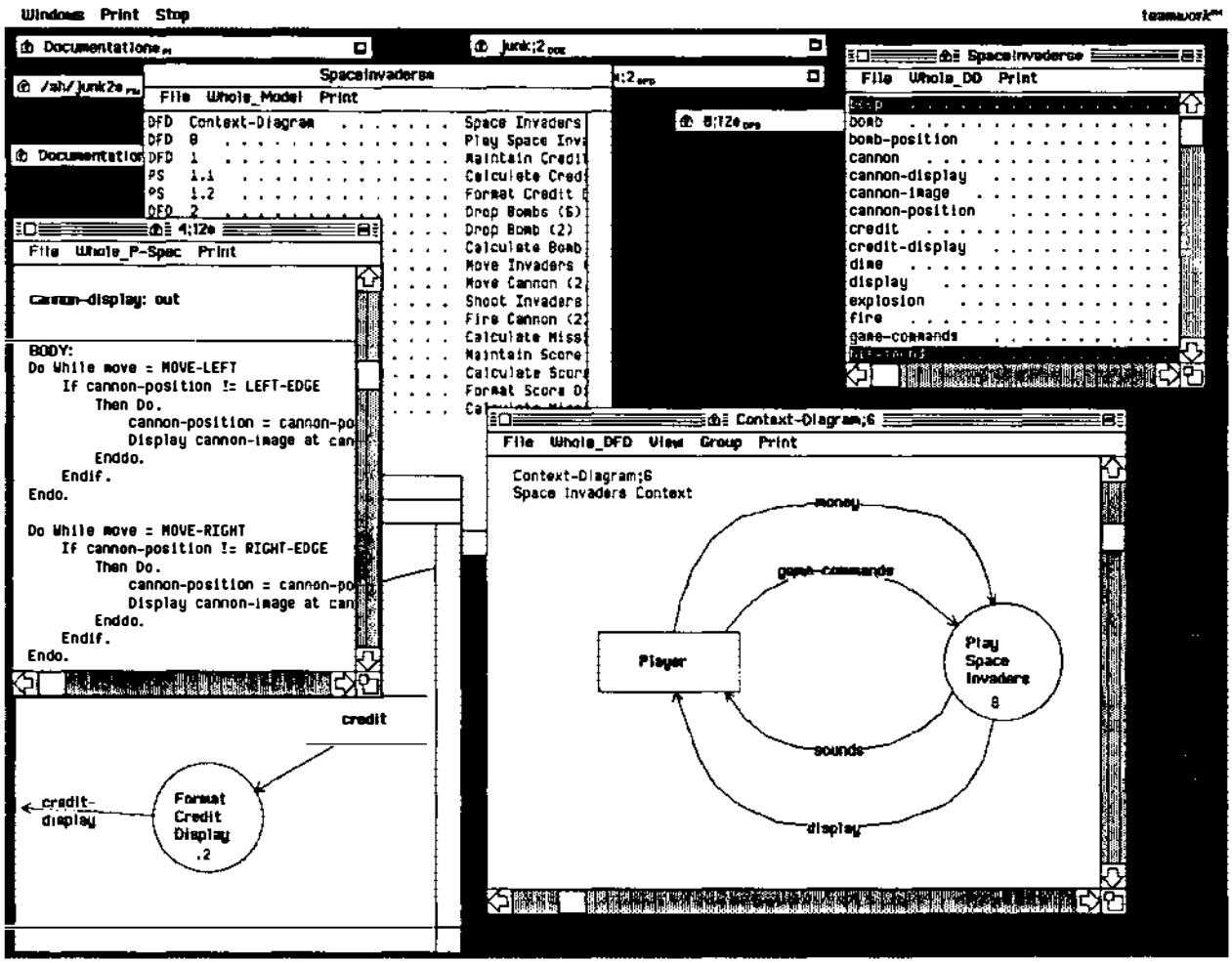



**Figure 4: team*work*/SA Desktop**

The graphics editors allow diagrams to be easily produced and edited. Diagrams as well as components of diagrams are automatically numbered and indexed. These features eliminate many manual, time consuming tasks.

Project information is retained in a project library, through which individuals can simultaneously share model information and computer resources. Team members linked over the network can access the same information for review. Multiple versions of model objects are retained in the library. Team members can independently renumber and repartition diagrams, which allows exploration of different approaches to describe a system.

**Teamwork/SA**'s consistency checker detects specification errors within and between data flow diagrams, data dictionary entries, and process specifications. Typical errors and inconsistencies include DFD balancing errors (data flows from one diagram that do not match data flows to a related diagram) and undefined data dictionary entries. The consistency checker uses the semantics and rules of structured analysis. Checking is performed "on-demand", which allows the analyst to perform analysis top-down, bottom-up, or any other way. It encourages the exploration of partial models that may be (during the intermediate stages of building the model) incomplete or incorrect. The speed and depth of checking in **teamwork/SA** helps produce consistent and correct specifications.

## Conclusions

The **teamwork/SA** software was developed using the structured methods described in this paper. This initial analysis was done in less than 3 man-years for the core of **teamwork/SA**. The only automated tools used were a text editor for entering and editing data dictionary entries and process specifications, and a filter for formatting the data dictionary entries. About 50 data flow diagrams, 120 process specifications, and 400 data dictionary entries were generated (these numbers do not include an analysis of the user interface). Even with the relatively small number of process specifications and data dictionary entries, checking the model became impossible except in a very localized manner. The software that implements the part of the system described above contains about 150,000 lines of code, and has less than 15 level one (highest severity) bugs. We attribute the low bug rate partially to the quality of the specification, and feel that even better code would have resulted had we had **teamwork** available during analysis. Additionally, we gained much insight into using the techniques on a large, multi-person project by bootstrapping; the same techniques were used manually that were automated.

Our preliminary findings with the users of **teamwork/SA** is that productivity has increased due to the reduction in manual effort as described in this paper. One user of **teamwork/SA** has determined that a productivity improvement of 48% was realized by analysts using the automated tool over performing analysis manually. Automatic checking also helps reduce overall development time: manual checking is not only time consuming, but unreliable, and automated checkers also help reduce the number of specification errors. The interactive graphics and text editing capabilities allow analysts time to explore several approaches rather than spending time erasing and redrawing models. This results in better quality systems since many more alternatives can be reviewed.

**Teamwork/SA** is the first in a family of products designed to support the entire software life cycle. We have also developed tools to support structured design [Page-Jones 80] (**teamwork/SD**™), extensions to analysis for modeling real-time systems (**teamwork/RT**™), and an open data base access product (**teamwork/ACCESS**™). Future enhancements will include information modeling. Ultimately, we plan to automate and integrate the entire software development life cycle.

# References


[Boehm 84] Boehm, Barry W.
Verifying and Validating Software Requirements and Design Specifications.
*Software* , January, 1984.

[Brady 85] Brady, James T.
A Theory of Productivity in the Creative Process.
In *Proceedings of the 1st International Conference on Computer Workstations*, pages 70-79.
November, 1985.

[Burger 84] Burger, R.M., et. al.
The Impact of ICs on Computer Technology.
*Computer* :88-95, October, 1984.

[DeMarco 78] DeMarco, Tom.
*Structured Analysis and System Specification.*
Yourdon Press, New York, 1978.

[Gane 79] Gane, Chris and Trish Sarson.
*Structured Systems Analysis: Tools and Techniques.*
Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1979.

[Page-Jones 80] Page-Jones, M.
*The Practical Guide to Structured Systems Design.*
Yourdon Press, New York, 1980.

[Ramamoorthy 84]
Ramamoorthy, C.V., et. al.
Software Engineering: Problems and Perspectives.
*Computer* :191-209, October, 1984.

[Ross 77] Ross, D. and R.E. Schoman Jr.
Structured Analysis for Requirements Definition.
*IEEE Transactions on Software Engineering* SE-3(1), January, 1977.

[Wulf 80] Wulf, W.A.
Trends in the Design and Implementation of Programming Languages.
*Computer* :14-23, June, 1980.

# TEAM SYSTEMS ANALYSIS:

## A Methodology For Automating Software Analysis And Design

**By: Albert F. Case, Jr.**
**Nastec Corporation**
**24681 Northwestern Highway**
**Southfield, MI 48075**
**(313)353-3300**

### ABSTRACT

"Computers are getting more and more powerful, but the programs that run them cannot keep up. Business and the Pentagon are taking aim at the problem."

-- John Paul Newport, Jr., "A Growing Gap in Software", FORTUNE Magazine, April 28, 1986

Software design techniques such as Structured Analysis and Design have been available since the 1970s; however, they have achieved neither widespread use within systems development organizations nor the productivity gains hoped for. Team Systems Analysis is an approach to software development which combines the rigor of structured analysis and design methodologies with two additional components: "Power tools" for software development and a workshop approach for developing system requirements. This approach can significantly reduce the lead time to implement new systems while increasing the quality of the software product.

*Mr. Case is Director of Consulting & Education Services for Nastec Corporation. His most recent book* Information Systems Development *was published in 1986 by Prentice-Hall, and his next book,* Team Systems Analysis *will be released in 1987. Mr. Case is a frequent speaker at national conferences, and has been elected to the ACM Lectureship Program.*

## 1. PROBLEM:

### SOFTWARE DEVELOPMENT PRODUCTIVITY

Software development is rapidly evolving from a craft or art form to an engineering discipline. Significant advances have been made over the past several years in both methodologies for designing software and the automated "power tools" to support analysts, designers and programmers. The motivating factor for these developments is software development productivity --the need for today's software developers to create more software, in a shorter time frame, with fewer defects, that more closely meets the needs of the software's end-users. However, as can be witnessed in the recent FORTUNE article, " A Growing Gap in Software" [Newport,86], it is evident that the software development community has not yet achieved its desired objectives in this area.

Why has progress towards extensive improvements in productivity been so slow in development? The predominant reason for this state of the practice is that software developers have not made a significant change in the way they build systems in the past ten years. Structured methodologies, developed in the 1970s are very narrowly implemented, and have not achieved the anticipated level of benefits. Furthermore, "power tools" for software analysis and design are still relatively new, and have not been widely deployed in the software development community.

### 1.1. The State of Software Development

Research conducted this year supports this contention. There are nearly 900 thousand professionals employed in the development of software in the United States (analysts, designers, programmers and software engineers). In the past five years, only about 165 thousand individuals have been trained in structured development techniques (less than 20% of the total), and only about 20 thousand practitioners actually follow the dictates of the methodologies with rigor and precision. The primary reason for this is the labor intensity and tediousness of the documentation requirements for these techniques, compounded by the fact that structured techniques, in themselves, do not produce operating computer programs. This causes a level of frustration in users which ultimately results in short-cutting the methodologies or complete abandonment.

The labor intensity and tediousness of these techniques has been significantly alleviated by the availability of "power tools" for analysis and design, collectively referred to as programmer/analyst workbench tools, which first became commercially available in 1981. Since that time, the number of tools vendors has grown from one to more than thirty. Yet, only about 9 thousand workbenches have been installed -- amounting to only one percent of the total population of software developers. Power tools for programming, however, such as interactive, full-screen program editors, debuggers and third and fourth generation languages which automate the programming process are used by nearly all software developers. The ratio of computer terminals to programmers today is nearly one to one -- a significant advance over the late 1960s and early 1970s when terminals were shared by large numbers of programmers.

It would appear then, that a major contributing factor to lower development productivity is the inconsistant application of formal, structured techniques, aggravated by the lack of deployment of tools.

### 1.2. Procedures for Development

Another reason for the slow growth in productivity improvement is the procedure by which data are collected and validated to create a system specification. Typically, analysts and designers interview system users, and then collate the various inputs into an analysis or design document. These interviews are usually conducted serially. At completion, the design document is then distributed to users for review and approval.

This procedure for data collection and analysis is not necessarily the optimum means for achieving user consensus on the system specification. The result is often disagreement among the users on the specification document, resulting in significant re-work. On occasion, the problems with the specification document are not discovered until after the system has been programmed, at which time problems are uncovered in system or user acceptance testing, or after final installation.

Major deficencies found at this late stage of software development tend to:

1. delay final implementation of the system, thereby deferring system benefits;

2. increase the cost of software development by causing additional work which could have been avoided by achieving consensus on the specification prior to implementation; and,

3. impare the software development organization's credibility with the user community.

Studies performed by ITT, IBM, TRW and Mitre indicate that nearly 40% of the cost of a system can be attributed to the removal of system defects, and that 45%-65% of these defects occured in system design [Rush,85]. These results can be traced directly to inadequate communication between the software designer and the end user.

## 2. SOLUTION:

### THE TEAM SYSTEMS ANALYSIS APPROACH

The requirements for increased software development productivity are:

- improvement of communications between software developers and their users;

- early detection of software design defects; and,

- an increase in the efficiency with which system specification documents are prepared, analyzed and corrected.

To meet these requirements, an approach to software development known as Team Systems Analysis (TSA) has been identified. The TSA approach combines improved procedures for requirements definition with techniques for quality software development coupled with a suite of tools to automate much of the labor intensive and tedious development tasks.

## 2.1. Improved Communications

To improve the communications between users and software developers, a number of formal procedures have arisen over the last several years which rely on group or team workshops for requirements data collection, rather than individual interviews alternated with group specfication reviews. These workshop oriented techniques gather users together with systems development representatives to discuss, in a highly structured fashion, the system requirements. The main objectives of these procedures are to:

- Achieve consensus among the users as to system requirements;
- Stimulate discussion among users and development personnel as issues arise; and,
- Formally document the results achieved.

These group procedures have been developed by a variety of firms and are commercially available. They include:

- JAD: Joint Application Design - IBM Corporation;
- FAST: Facilitated Application Specification Technique - MG Rush Systems, Inc.;
- CONSENSUS - Boeing Computer Services, Inc.;
- THE METHOD - Performance Resources, Inc.;
- ASAP: Accellerated Sytems Analysis Procedures - Dan Myers and Associates, Inc.;
- WISDM: WISE Information Systems Development Methodology - Western Institute of Software Engineering, Inc.;

For many of these techniques, the vendors will train individuals within the user or systems development community as facilitators for the workshops, others perform workshop facilitation as a consulting service.

The benefits of these formal procedures fall in two areas: Reduced calendar time to complete the system development specification and a significant reduction in changes required in the specification as development progresses. Boeing Computer Services claims a 60% to 80% reduction in time required to perform systems analysis using CONSENSUS and an overall 50% savings in time for the entire development effort.

### 2.2. Early Detection of Errors

Once a specification has been developed, it must be validated for accuracy. Structured analysis and design techniques, such as those developed by Yourdon, Demarco and Gane & Sarson provide a set of rules for developing the specification and analyzing it for accuracy and completeness.

The primary benefits of structured techniques are: reduced development time and cost, and reduced resource requirements for system maintenance. This is possible because these techniques provide:

- Unambiguous specifications which are less likely to be mis-interpreted in the implementation process;
- Rules for validation which ensure accuracy and completeness of the specification; and,
- Increased level of abstraction of the problem and solution.

The structured techniques provide a language or syntax for the development of systems specifications, much like programming languages establish the syntax for machine instructions. This syntax can then be validated by using rules established by the technique -- much like a compiler automatically, or a programmer manually, checks the syntax of the program.

Other major productivity improvement which can be obtained through the application of a structured technique are redundancy elimination and reusability. By rigorusly defining system components in a standard format, the specification can be searched for processes or programs which perform similar or identical functions, a task that is very difficult, if not impossible, with a non-structured specification.

### 2.3. Increasing Efficiency

The process of specifying systems using the TSA approach, entails the combination group data collection procedures and structured specification techniques to document the results.

As previously mentioned, the documentation of the specification and the subsequent validation and analysis are labor-intensive and time consuming processes. Unlike programming, which applies the power of the computer to such mundane tasks as checking source code syntax, relatively few developers are using automated tools to develop and analyze system specifications. Using an analyst workbench tool, the initial time required to create a specification document can be reduced by 15% to 20%.

Even more dramatic productivity improvements evolve as specification validation begins. A recent study [Jones,86] indicated that as much as 30% of the code in present systems is redundant. The prime reason for this redundancy is the difficulty of employing manual procedures to identify redundant or reusable modules. However, once the structured specification has been reduced to data base entries, such searches can be conducted automatically through data base queries.

Reducing the specification to a data base also enables such validation tasks as verification of design components, cross referencing and syntax checking can be performed by software rather than by people.

These automated facilities, currently available in commercial products, can further decrease the resource required to perform analysis and design by and additional 40% to 60%. Since the specification is in machine readable form, some products offer export facilities which will translate the specification into input for an application generator which can virtually eliminate programming (and hence human programming errors) once the specification is complete.

### 2.4. Summary

Team Systems Analysis is an approach to developing software systems which combines a workshop approach to collecting requirements information from users, documenting those requirements using a structured analysis and design methodology, and employing analyst/ programmer workbench tools to aid in the data capture and analysis.

The remainder of this paper is devoted to a description of the workshop approach to collecting requirements and the workbench requirements to support the documentation and analysis tasks.

## 3. THE PROCEDURE:

### THE WORKSHOP APPROACH TO REQUIREMENTS DEFINITION & ANALYSIS

How do workshop approaches to system analysis and requirements definition differ from more conventional approaches to analysis and design? First, users are congregated into workshop sessions which replace the traditional user interviews. Unlike previous "design by committee" approaches, however, the workshop methodologies have rigid agendas, highly structured data collection procedures and roles to be carried out during the workshop sessions.

### 3.1. Workshop Roles

The workshop sessions are populated predominantly by users -- both management and professional personnel responsible for providing data to the system, and using its results. In addition to the users, the other participants in the workshop session are the facilitator, the software engineer and systems consultants.

#### THE FACILITATOR

The facilitator is responsible for directing workshop sessions, leading discussions by asking pre-determined questions about system requirements and "coaching" the discussions to ensure that all of the relevant requirements information is captured. The facilitator must have a working knowledge of systems development and the technology to be employed in the development of the system, as well as knowledge of the users needs.

Most consultants in this field recommend that the facilitator be an impartial party who has vested interests in neither the systems development organizaton nor user areas. Some consulting firms provide facilitator training while others provide facilitators on contract to the organization. Facilitators who are not familiar with the organization and the user departments conduct a preliminary fact- finding analysis prior to conducting the workshop sessions.

#### THE SOFTWARE ENGINEER

The software engineers (also referred to a "scribes" in some methodologies) are responsible for the actual collection and compilation of data from the workshop sessions. Many of the workshop methodologies provide pre-printed forms for the collection of data from the sessions while others combine structured analysis and design techniques with forms. The data collection during the sessions takes the form of working papers which are summarized by the software engineers into a final specification document.

In addition to the collection of data from the sessions, the software engineers are often responsible for validation of the data collected -- such as identification of redundant processes and the assurance of consistancy in naming conventions.

Since the software engineer must be able to translate the discussions ensuing during the workshop to a specification, the individual needs strong systems development background, and must be trained in a formal specification technique. In addition, training in the specific workshop methodology may be required.

### THE SYSTEMS CONSULTANT

The systems consultants represent the systems development organization during the analysis and requirements definition workshops. Their participation is typically limited to commenting on the technical feasibility of particular implementation approaches. The systems consultants who participate in the workshop sessions are the same personnel who will be responsible for the implementation of the system. So, in addition to providing input to the workshop sessions, the consultants also are thoroughly familiar with the requirements. The consultant roles are often filled by the systems development project manager and one or more senior analysts who will participate in the project.

### THE USERS

The users are central to the success of the workshop sessions, for they are the decision makers in a consensus-oriented process. In determining the participants in the session it is a requirement that decision makers from each represented user area be present. In addition, knowledgeable personnel at every level who can make a significant contribution to requirements are invited. Participation by the users is essential since they make all of the decisions. The systems consultants merely advise the participants.

While formal training for users is most often not required, most workshop methodologies include a detailed orientation to the process at the start of the workshop sessions.

## 3.2. The Workshop Process

While all of the workshop methodologies share many similarities, they also differ in the detailed procedures for handling the workshop sessions and the pre-session preparation. IBM's Joint Application Design, for example, takes a fairly traditional approach to the method for documenting system requirements, while other workshop methodologies follow the structured analysis and design tasks as outlined by Yourdon [79] and Demarco [79], augmenting these procedures with a controlled workshop environment. A typical workshop project plan would include the following activities:

- ACTIVITY 1: INITIATE THE PROJECT
- ACTIVITY 2: ORIENT THE FACILITATOR
- ACTIVITY 3: CONDUCT THE SYSTEMS ANALYSIS WORKSHOP
- ACTIVITY 4: CONDUCT REQUIREMENTS DEFINITION WORKSHOP
- ACTIVITY 5: DOCUMENT THE FINAL RESULT
- ACTIVITY 6: TRANSITION TO DESIGN AND IMPLEMENTATION

A detailed work breakdown structure for a project is include in the Appendix to this paper.

## PROJECT INITIATION

Project initiation begins by the requestor writing a synopsis of why a software development project needs to be undertaken:

- What is the problem or opportunity?
- Why is the current system insufficient?
- What additional information/processing is required?

Once this preliminary synopsis has been developed, the process of planning the analysis and requirements definition project can begin.

Users are the key to the workshop process, and it is essential that the appropriate users attend the workshop sessions. The first task is to identify participants. Appropriate user participants are those who:

- Make decisions based on the information from the system
- Control the funding for the system's development
- Directly interface with the system at the report, input or inquiry level.

## ORIENTING THE FACILITATOR

Whether the facilitator is an employee or a consultant, it is necessary to orient the user to the problem. This activity usually entails a pre-workshop process where the facilitator collects and reviews current system documentation, copies of source documents and reports, and conducts some preliminary interviews with the project requestors.

Based upon this activity, the facilitator can then plan the workshop sessions and prepare some initial presentation materials. The facilitator should have collected sufficient data to prepare an initial Context Level data flow diagram describing the present automated or manual system.

## CONDUCT THE SYSTEMS ANALYSIS WORKSHOP

The primary objective of the Systems Analysis workshop is to validate and expand the model of the current system prepared by the facilitator during orientation. The resulting specification from this activity is a Current Physical Model [DeMarco,79] which is agreed to by the users. This model is then used to identify portions of the system which must be automated or modified based upon the users' requirements.

During this session various data flow diagrams, input documents and output reports (screens and hardcopy) are presented to the group of assembled users. Each input and output is discussed until a generally accepted definition is achieved. During the workshop, the software engineer(s) update dictionary definitions and diagrams to reflect the consensus of the group.

## CONDUCT THE REQUIREMENTS DEFINITION WORKSHOP

To this point, discussions have focused on the nature of the current system, with little or no discussion of why the system must be changed. Now that a complete model of the system, agreed to by the users, and reflecting the way the system actually works, is available, the process of identifying required changes to the system begins.

Using the current system model and the revised problem statement as a guideline, the facilitator formulates a series of questions to elicit system requirements from the users. When the consensus of the group is achieved, these requirements, documented by the software engineer, represent the changes which must be made to the system.

In addition to narrative statements, these requirements can take the form of new or modified report and screen layouts and modified data flow diagrams. As this process continues, the software engineer continually updates the design dictionary.

## DOCUMENT THE FINAL RESULT

Throughout the workshop process, the users are achieving consensus on the system specification, while the software engineer is documenting the results (usually on pre-printed forms). This documentation is then assembled into a final specification which is distributed to the participants for review and final approval. If there are any requested changes to the final document, the group is convened for a final discussion of the changes. Once the specification document has been finalized, it is turned over to the team of developers responsible for the design and implementation of the system. The final specification document is outlined as follows:

- Problem Statement
- Current System Physical Model
    - Context Level DFD
    - Lower Level DFD's
    - Process Narratives
    - Current System Design Dictionary
        - Object Definitions
        - Process and Data Structures
        - Cross Reference Listings
        - Object Relations
- Current System Logical Model
- Requirements Statements
- Proposed System Logical Model
- Report & Screen Layouts

From this document, the implementation team then develops the Proposed System Physical Model, process specifications, programs and user procedures.

### 3.3. Rules for Running the Sessions

Users, like software developers, are backlogged with work, and find it difficult to take several consecutive days out of their schedules to participate in the workshops. There are several ways to gain their participation.

First, it should be pointed out that, while it will require several contiguous days of participation, the result will be far less disruptive than numerous occasional interviews and design reviews, which will be inevitably followed up by additional meetings to correct misunderstandings or shortcomings in the system specification.

Second, the probability that the system will meet the users' needs is substantially enhanced. Since the workshops are a structured dialog among the systems users, ideas, problems, concerns and implementation strategies can be discussed in a group.

Third, by assembling the users and obtaining consensus through the workshops, the time from project request to system operation will be substantially reduced.

Occasionally, it is necessary to resort to "friendly persuasion" to assure participation. A letter from the senior user indicating that participation is required is often effective. Another alternative is for the systems development organization to indicate that this is a process for negotiating a "contract" for the system. Only changes requested by participants will be accepted.

#### GROUND RULES

The workshop process is highly democratic. In the book *Information Systems Development* [Case, 86, p.209], the following rules were established as a guideline for working relationships between systems development and users:

1. The user community and systems development will always be honest and forthright in their communications with each other.

2. The user has the absolute right to state what he or she wants from the new system.

3. The user has the absolute right to state how much can be spent on development and implementation.

4. Systems development has the responsibility to seek the lowest-cost method to implement the entire system, determined by quantifiable, documented techniques, and to inform the user of the true cost, whatever it may be.

5. DP will never say "It can't be done" unless they mean it. That phrase is *not* synonymous with "It takes more time than you want" or "It will cost more than you allocated" .

6. If the cost [or time] to build exceeds the funding allocated, the users must accept reduced functionality or provide more resources.

7. Users will never say "I want it all for less money/in less time" .

These rules apply equally well to the workshop sessions, with some minor additions:

8. The role of the systems consultant is to advise on feasibility, cost and schedule, not to make decisions on requirements. The systems consultants have no vote on requirements or logical design.

9. The facilitator never has an opinion.

10. The specification cannot be changed without a meeting of the workshop participants, no matter how trivial the change may appear.

### 3.4. Physical Meeting Requirements

In addition to a "code of conduct" for the workshop sessions, it is essential to have a proper physical environment for the meetings. This includes the necessary tools, and accommodations.

#### ACCOMMODATIONS

Each workshop session can last from two to five days. During that time, the participants should be isolated from the day to day operations of the organization. A meeting facility off-site is preferable. The facility should have sufficient lighting, seating and work space.

Typically, these workshops are intensive sessions. Often, in an attempt to minimize calendar time devoted to the project, sessions run far longer than eight hours. The net effect of longer days is clouded thinking, which may have the impact of requiring additional session days. Rather, it is preferable to run 8 hour days with at least an hours worth of breaks, keeping the working sessions to 6 hours.

#### TOOLS

**The workshop facility should be equipped with:**

**2 marker boards**
**2 flip-chart easels**
**wall space for hanging sheets**
**2 overhead projectors**

Additional requirements, for an automated session would include a personal computer, a large-screen projection unit. If more than one software engineer is in the session, the two PCs should be linked to a common data base via a local area network.

## 4. THE TOOLS:

### COMPUTER-AIDED SOFTWARE ENGINEERING TECHNOLOGY

The productivity gains which can be achieved by providing the software engineers with Computer-Aided Software Engineering (CASE) systems for the capture and analysis of information collected during the workshop sessions. CASE systems allow the software engineers to:

- Capture design information on-line, rather than on pre-printed forms
- Interactively check the definition of data elements and processes during the session
- Develop prototype screens and reports on-line
- Update definitions, diagrams and layouts as changes occur
- Validate and balance data flow diagrams during the sessions so that discrepancies can be discussed with the users as they occur
- Automatically track requirements against proposed system models to ensure that no requirements go unaddressed.
- Provide a machine readable specification document to the implementation team, and ultimately, to an application generator.

This results in a higher quality specification document which results in fewer sessions required to "iron out" specification changes, and significantly reduces development time by having a completely validated specification, free of design errors. Also, these systems enable the developers to automatically identify redundant or reusable modules which can reduce the total development effort by as much as thirty per cent [Jones, 86]. By capturing the specifications on-line, with a CASE system, as much as 90% of the code can generated from the machine readable specification document [Freedman, 86].

### 4.1. CASE System Requirements

The real question is which CASE system to employ. Since 1981, when the first commercially available analyst/programmer workbench came on the market, the number of vendors has grown to over 30, with prices ranging from $995 to $9,000. The range in product functionality is equally broad. Listed below are desirable characteristics of a tool which would be adequate to support a workshop session:

1. FORMS: Most workshop methodologies provide a plethora of pre-printed forms for the capture of data. The CASE system would ideally have the forms available on-line, or at a minimum, the means to design and store forms in an on-line forms library.

2. SHARABLE DATABASE: Often, a session will have more than one software engineer. In that case, the tool selected should allow multiple workstations to interactively access a single data base. Since, quite often, the systems will be off-site, the ability to share a mainframe data base is highly desirable.

3. GRAPHICS WORD PROCESSING: Many of the documents created during these sessions are a mixture of narratives and graphics. A system with a graphics-text editor is an advantage. Graphics support should include:

   - Data flow diagramming
   - Entity/Relation modelling
   - Matrix development
   - Multiple methodology support

4. DESIGN DICTIONARY: The dictionary data base should be highly extensible allowing the logging of object definitions, where used occurences and how-used relations. The relations stored in the dictionary should allow the formulation of inquiries such as "What processes transform input X into output Y ?", "List potentially redundant processes." The dictionary should be accessible directly from either word processing or diagramming operations.

5. ANALYTICAL SUPPORT: Many analyst functions such as data flow diagram validation and data flow balancing can be off-loaded to the machine. Systems which support these functions are essential. In addition, the system should automatically:

   - Populate the dictionary with definitions based on usage in a diagram, if no definition is found. This feature should also be on a toggle, since automatic population is not always desirable.

   - Catalog where-used cross reference information for diagrams and narratives

   - Extract "how used" relations from diagrams.

6. CODE GENERATOR INTERFACE: The process of specifying programs begins with analysis and requirements definition. Often, after a structured specification has been developed, the information must be re-entered, in another format to a program generator (or, worst case to a compiler). Since the specification is captured in a data base, when using a CASE system, it is now possible to have the CASE data base automatically input to the generator, with only the incremental information being added.

## ADDITIONAL REQUIREMENTS

In addition to those requirements listed, which are specifically designed to support the workshop sessions, additional characteristics are desirable in the tool and the vendor.

User interface is one of the most common decision factors with respect to tools. Most of the workbench products have "Mac-like" interfaces incorporating mouse usage. The user interface should provide:

- A beginner-mode with pop-up menus and leading prompts for the first-time or occasional user.
- An expert mode which avoids the menus for the expert user
- An on-line Help function

One factor to consider in the user interface however, is the functionality of the product. The more features a product has, the more commands and menu choices will be available, hence, a longer learning curve. Provided that the user-interfaces meet the minimum requirements, the trade-off between a simpler system which takes 1 day to master, and a more powerful system which takes 2 or 3 days to master, the choice for the more powerful system will pay-off over time. The two extra days of adjustment are insignificant over the life of a project, or the tool.

## TRAINING

Having a power saw does not one a carpenter make. The saw, in skilled hands, simply gets a quality product faster. Likewise, in unskilled hands, it simply gets a kludge more quickly. The same analogy holds true for analysis tools. They are not a substitute for a knowledge and understanding of structured techniques. They simply off load the rather simple, but bothersome and time consuming tasks to a machine. Therefore, it is essential that the software engineers obtain the necessary proficiency in the design techniques. Some vendors offer training in the structured techniques, which combine theoretical training with hands-on workshops using the tool. This could be a significant factor in the selection of a tools vendor.

## 5. SUMMARY

The Team Systems Analysis approach is a combination of proven design techniques and tools, coupled with newer, more efficient means of communicating with users. The net result is faster system implementation, fewer specification changes, and a higher quality system.

Although the concept of integrating workshop methodologies with structured techniques and CASE tools is relatively new, there has been demonstrable success in the commercial environment. One such example is Carrier Corporation, a division of United Technologies. Carrier undertook a project using a Team Systems Analysis approach, and the results were published in *Information Center* magazine [Salm,86].

Carrier, already a user of structured analysis and design, adopted IBM Corporation's Joint Application Design (JAD) workshop methodology, Nastec Corporation's DesignAid analyst/programmer workbench, and the Telon application system generator from Pansophic. The JAD procedures, modified by Carrier's development organization, and data capture forms were loaded into DesignAid's data base. Armed with this development environment, they began a project to build a system consisting of 12 IMS/DC transactions and interfaces to 3 batch applications. The total project was originally estimated at 5153 hours. Carrier's estimating techniques are quite sophisticated, using development statistics dating since 1977. The 3 year history (1982 - 1985) of project completions showed that targets were hit within 10% of estimate consistantly.

The results of the project were impressive. Their results showed an improvement in requirements and design productivity of 37%, and an overall project productivity increase of 45%. In an interview with a former Carrier manager, subsequent to the publication of the article, he indicated that, there was no maintenance required for 9 months after project completion.

### IN CONCLUSION ...

The Team Systems Analysis approach is a combination of proven design techniques, coupled with a new generation of tools and a modern method of communicating with users. The net result is faster system implementation, fewer specification changes, and a higher quality system.

### APPENDIX: THE WORK BREAKDOWN STRUCTURE

*ACTIVITY 1.1 INITIATE THE TEAM SYSTEMS ANALYSIS PROJECT*

TASK 1.1.1 DEVELOP WRITTEN PROBLEM STATEMENT

TASK 1.1.2 SET UP PROJECT

WORKSTEP 1.1.2.01 Identify key user personnel
WORKSTEP 1.1.2.02 Identify key systems development personnel
WORKSTEP 1.1.2.03 Assign Team Systems Analysis team members

WORKSTEP 1.1.2.04 Develop Phase 1 Action Plan
WORKSTEP 1.1.2.05 Send out Team Notices

*ACTIVITY 1.2 ORIENT THE FACILITATOR*

TASK 1.2.1 DEFINE THE ORGANIZATION

WORKSTEP 1.2.1.01 Develop a user organization chart
WORKSTEP 1.2.1.02 Catalog user organizational units
WORKSTEP 1.2.1.03 Define user organization responsibilities
WORKSTEP 1.2.1.04 Define organizational unit responsibilities
WORKSTEP 1.2.1.05 Define participant job descriptions
WORKSTEP 1.2.1.06 Define user systems
WORKSTEP 1.2.1.07 Define user system interfaces
WORKSTEP 1.2.1.08 Define intradepartmental communications
WORKSTEP 1.2.1.09 Define interdepartmental communications
WORKSTEP 1.2.1.10 Define external communications

TASK 1.2.2 DEVELOP CURRENT PHYSICAL MODEL

WORKSTEP 1.2.2.01 Collect existing system documentation
WORKSTEP 1.2.2.02 Document user explanation of system functionality
WORKSTEP 1.2.2.03 Review existing system documentation
WORKSTEP 1.2.2.04 Document inconsistancies
WORKSTEP 1.2.2.05 Document system transactions
WORKSTEP 1.2.2.06 Define sources of data
WORKSTEP 1.2.2.07 Define destinations of data
WORKSTEP 1.2.2.08 Develop Context Level data flow diagram
WORKSTEP 1.2.2.09 Develop System Process Narrative
WORKSTEP 1.2.2.10 Develop data structures
WORKSTEP 1.2.2.10 Develop Level 0 data flow diagram
WORKSTEP 1.2.2.11 Develop Level 0 process narratives
WORKSTEP 1.2.2.12 Balance Context & Level 0
WORKSTEP 1.2.2.13 Document balancing errors

TASK 1.2.3 UNDERSTAND THE PROBLEMS / OPPORTUNITIES

WORKSTEP 1.2.3.01 Schedule Facilitator Orientation Consensus Session
WORKSTEP 1.2.3.02 Review written User Problem Statement
WORKSTEP 1.2.3.03 Identify involved system outputs
WORKSTEP 1.2.3.04 Document recommended changes to system outputs
WORKSTEP 1.2.3.05 Identify involved processes in the current system
WORKSTEP 1.2.3.06 Develop Facilitator Problem Statement
WORKSTEP 1.2.3.07 Define the scope of the systems project

TASK 1.2.4 ASSEMBLE FACILITATOR ORIENTATION REPORT

TASK 1.2.5 CONDUCT FACILITATOR ORIENTATION CONSENSUS SESSION

WORKSTEP 1.2.5.01 Review & Revise TSA objectives
WORKSTEP 1.2.5.02 Review & Revise User Problem Statement
WORKSTEP 1.2.5.03 Review & Revise Context Diagram
WORKSTEP 1.2.5.04 Review & Revise Facilitator Problem Statement

*ACTIVITY 1.3 CONDUCT THE SYSTEMS ANALYSIS WORKSHOP*

TASK 1.3.1 SCHEDULE SYSTEMS ANALYSIS WORKSHOP

WORKSTEP 1.3.1.01 Identify systems analysis workshop participants
WORKSTEP 1.3.1.02 Schedule systems analysis workshop dates
WORKSTEP 1.3.1.03 Reserve conference facilities & equipment
WORKSTEP 1.3.1.04 Write & deliver meeting notices

TASK 1.3.2 PREPARE FOR SYSTEMS ANALYSIS WORKSHOP FACILITATION

WORKSTEP 1.3.2.01 Review Facilitator Orientation Report
WORKSTEP 1.3.2.02 Develop agenda
WORKSTEP 1.3.2.03 Create overheads
WORKSTEP 1.3.2.04 Create slides

TASK 1.3.3 PRESENT SYSTEMS ANALYSIS WORKSHOP INTRODUCTION

WORKSTEP 1.3.3.01 Present objectives of Systems Analysis Workshop
WORKSTEP 1.3.3.02 Present Agenda
WORKSTEP 1.3.3.03 Present revised Facilitator Problem Statement
WORKSTEP 1.3.3.04 Discuss & Modify Facilitator Problem Statement

TASK 1.3.4 REVIEW CONTEXT OF APPLICATION SYSTEM

WORKSTEP 1.3.4.01 Review Context Level data flow diagram
WORKSTEP 1.3.4.02 Review/correct each system input
WORKSTEP 1.3.4.03 Review/correct each system ouptut
WORKSTEP 1.3.4.04 Review/correct each system input data structure
WORKSTEP 1.3.4.05 Review/correct each system output data structure
WORKSTEP 1.3.4.06 Review/correct Context Level Process Narrative

TASK 1.3.5 REVIEW LEVEL 0 OF APPLICATION SYSTEM

WORKSTEP 1.3.5.01 Review Level 0 data flow diagram
WORKSTEP 1.3.5.02 Review/correct each Level 0 input
WORKSTEP 1.3.5.03 Review/correct each Level 0 ouptut
WORKSTEP 1.3.5.04 Review/correct each Level 0 input data structure
WORKSTEP 1.3.5.05 Review/correct each Level 0 output data structure
WORKSTEP 1.3.5.06 Review/correct Level 0 Process Narratives
WORKSTEP 1.3.5.07 Balance Context and 0 Level diagrams

TASK 1.3.6 OBTAIN USER CONSENSUS SIGN-OFF FOR SYSTEMS ANALYSIS

WORKSTEP 1.3.6.01 Prepare final Systems Analysis Workshop Report
WORKSTEP 1.3.6.02 Distribute Systems Analysis Workshop Report
WORKSTEP 1.3.6.03 Solicit approval of Systems Analysis Workshop Report
WORKSTEP 1.3.6.04 Arbitrate changes to be made to final report

*ACTIVITY 1.4 CONDUCT REQUIREMENTS DEFINITION WORKSHOP*

TASK 1.4.1 SCHEDULE THE REQUIREMENTS DEFINITION WORKSHOP

WORKSTEP 1.4.1.01 Identify Definition Workshop participants
WORKSTEP 1.4.1.02 Schedule Requirements Definition Workshop dates
WORKSTEP 1.4.1.03 Reserve conference facilities & equipment
WORKSTEP 1.4.1.04 Write & deliver meeting notices

TASK 1.3.2 PREPARE FOR REQUIREMENTS DEFINITION WORKSHOP FACILITATION

WORKSTEP 1.4.2.01 Review Facilitator Orientation Report
WORKSTEP 1.4.2.02 Review Systems Analysis Workshop Report
WORKSTEP 1.4.2.03 Develop agenda
WORKSTEP 1.4.2.04 Create overheads
WORKSTEP 1.4.2.05 Create slides

## SUGGESTED READINGS


BOEHM, BARRY W., *Software Engineering Economics*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1981.

CASE, ALBERT F., "Computer-Aided Software Engineering (CASE): New Technologies for Systems Development Productivity," *Database Magazine*. Quarterly publication of the Association for Computing Machinery, Special Interest Group for Business Data Processing, Winter, 1986.

CASE, ALBERT F., *Information Systems Development: Principles of Computer-Aided Software Engineering*. Englewood Cliffs, N.J.: Prentice-Hall, 1986.

DEMARCO, THOMAS, *Structured Analysis and System Specification*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1978

FREEDMAN, DAVID H., "Programming without Tears", *High Technology Magazine*, April,1986. p. 38.

FREEDMAN, DAVID H., "Workstation Software: Does It Go Far Enough?", *Infosystems*. April,1986. p. 68.

JONES, T. CAPERS, *Programming Productivity*. New York: McGraw-Hill Book Company, 1986.

MANLEY, JOHN H., "Computer-Aided Software Engineering (CASE): Foundation for Software Factories", IEEE COMPCON '84 CONFERENCE ON THE SMALL COMPUTER (R)EVOLUTION PROCEEDINGS, September 16-20, 1984 (Silver Spring, MD: IEEE Computer Society Press, 1984).

NEWPORT, JOHN PAUL JR., "A Growing Gap in Software", *Fortune Magazine*. Vol. 113, No. 9, April 28, 1986. pp. 132-142.

ORR, KEN, *Structured Requirements Definition*. Topeka, Kans.: Ken Orr & Associates, Inc., 1981.

PRESSMAN, ROGER S., *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill Book Company, 1982.

RUSH, GARY, "A Fast Way to Define System Requirements", *Computerworld*. October 7, 1985. pp. ID/11-ID/75.

SALM, GARY, "The Software Engineering Approach to Application Development," *Information Center*. Vol. II, No. 4, April,1986. pp. 38-43.

YOURDON, EDWARD AND LARRY CONSTANTINE, *Stuctured Design: Fundamentals of a Discipline of Computer Program and System Design*. New York: Yourdon Press, Inc., 1978.

# TEAM SYSTEMS ANALYSIS




1

## THE STATE OF SOFTWARE DEVELOPMENT

- **Use of Structured Techniques**
  - **< 20% of population**
  - **Complex**
  - **Time consuming**
- **CASE Tools**
  - **Used by < 1%**

2

## TSA GOALS

- Improve communications between development and end users
- Early detection of defects
- Reduce development time



## TSA METHOD

- Improve Communications
  - Consensus
  - Free flow of information
  - Formally documented results

## TSA METHOD

- Methodologies
  - JAD
  - Fast
  - Consensus
  - The Method
  - ASAP
  - WISEDM



## TSA METHOD

- Early Detection of Errors
  - Unambiguous
  - Validation rules
  - Abstraction

## TSA METHOD

- Increased Efficiency
  - Automate manual tasks
  - Redundancy elimination
  - Reusability



## THE PROCEDURE

- A Workshop Approach
  - Roles

## THE PROCEDURE

- Workshop Roles
  - Facilitator
  - Software Engineer
  - Systems Consultant
  - Users



## THE PROCEDURE

1. Initiate the project
2. Orient the facilitator
3. Analysis workshop
4. Requirements workshop
5. Document consensus
6. Transition to design

## RULES

1. The user community and systems development will always be honest and forthright in their communications with each other.

2. The user has the absolute right to state what he or she wants from the new system.

3. The user has the absolute right to state how much can be spent on development and implementation.



## RULES (cont.)

4. Systems development has the responsibility to seek the lowest-cost method to implement the entire system, determined by quantifiable, documented techniques, and to inform the user of the true cost, whatever it may be.

5. DP will never say "It can't be done" unless they mean it. That phrase is <u>not</u> synonymous with "It takes more time than you want" or "It will cost more than you allocated" .

6. If the cost [or time] to build exceeds the funding allocated, the users must accept reduced functionality or provide more resources.

## RULES (cont.)

7. Users will never say "I want it all for less money...in less time."

These rules apply equally well to the workshop sessions, with minor additions:

8. The role of the systems consultant is to advise on feasibility, cost and schedule, not to make decisions on requirements. The systems consultants have no vote on requirements or logical design.





## RULES (cont.)

9. The facilitator never has an opinion.

10. The specification cannot be changed without a meeting of the workshop participants, no matter how trivial the change may appear.

## THE PROCEDURE

- Accommodations
  - 2 to 5-day time frame
  - Preferably off-site
  - 8-hour limit

## WORKSHOP ROOM LAYOUT

1. 6-foot projection screen
2. 6-foot projection screen
3. Facilitator's PC
4. PC projector
5. Overhead projector
6. U-shaped work area
7. Scribe PC
8. 35mm projector

## THE TOOLS

- Computer-Aided Software Engineering



## THE TOOLS

- Objectives
  - Increase data-capture speed
  - On-line reference
  - Prototyping
  - Validation
  - Machine-readable spec.

## THE TOOLS

- CASE System Requirements
  - Forms
  - Shareable data base
  - Dictionary
  - Analytical support
  - Code generator interface



## THE TOOLS

- Additional Features
  - Multiple modes
  - On-line help

## THE TOOLS

- Training
  - Workshop
  - Methodology
  - Tool
  - Integration



## RESULTS

- Reduced time and resources
- Reduction in programming and testing
- Reduced software maintenance
- Meet user requirements

## CASE STUDY

- Carrier Corporation
  - DesignAid (NASTEC)
  - JAD (IBM)
  - TELON (PANSOPHIC)



## CASE STUDY

- Results
  - 45% overall productivity improvement
  - 37% in design
  - 79% decrease in analyst time - test.
  - No maintenance for 9 months

## CASE STUDY

- Results based on initial estimates
- Estimates from historical data
- Track record ± 10%, last 3 yrs.



## CONCLUSIONS

- Structured methods are valuable tools for quality.
- Workshop methodologies improve user communication.
- CASE tools make methods feasible.
  - Formally documented results

# EXPERIENCE USING A STATIC ANALYZER TO IMPROVE SOFTWARE QUALITY IN A LARGE SYSTEM


## ABSTRACT

Programmers who analyze or maintain large software programs must deal with a series of complex relationships among program elements, module interfaces, and logic flow. This paper describes the author's experience using a static analyzer to analyze one large production Fortran system. A static analyzer is defined, and the features and characteristics of a specific static analyzer, the Maintainability Analysis Tool (MAT), are described. Details of errors and poor programming practices detected by MAT in the software system are presented, and two recurrent major patterns of problems are discussed. Recommendations for better use of Fortran are made.



**Lawrence Aaron McCarter**

Mr. McCarter is a Senior Software Engineer at Science Applications International Corporation in Arlington, Virginia. He specializes in the evaluation, testing, and improvement of application software packages.

Mr. McCarter received a B. S. in Mechanical Engineering from New Mexico State University, Las Cruces, N. M. in 1968 and a M. S. in Computer Systems from The American University, Washington, D. C. in 1974.

He is a member of the Association for Computing Machinery (ACM) and the Digital Equipment Computer Society (DECUS).

Lawrence A. McCarter
Science Applications International Corporation
1213 Jefferson Davis Highway
Suite 1500
Arlington, Virginia 22202
(703) 979-5910

# EXPERIENCE USING A STATIC ANALYZER TO IMPROVE SOFTWARE QUALITY IN A LARGE SYSTEM

Lawrence A. McCarter

November 1986

(Paper presented at the Fourth Annual Pacific Northwest Software Quality Conference, November 1986, Portland, Oregon)

Science Applications International Corporation
1213 Jefferson Davis Highway
Suite 1500
Arlington, Virginia 22202

# EXPERIENCE USING A STATIC ANALYZER TO IMPROVE SOFTWARE QUALITY IN A LARGE SYSTEM

## INTRODUCTION

Large software programs (those having in excess of 100,000 lines of high-level code) present special problems to those who implement, debug, test and validate them. The very size of such programs introduces whole sets of major, non-linear problems. Large software programs normally consist of separately compilable sets of code (modules or subprograms) linked together. As the number of subprograms constituting the program increases, the complexity of the logic, interfaces, subprogram relationships, and element name usages can quickly overwhelm even the most intelligent and experienced programmers.

In the practical world of applications software, it is not uncommon for a module to consist of from 25 to 1000 statements and contain from 15 to 2000 unique element names. A large software program may have more than 1,000 subprograms, with 10,000 or more uniquely defined element names (varibles, arrays, parameters, common block names, argument list names). One system with which we are involved has better than 258,000 lines of executable Fortran code, divided into 3,600 modules with 38,000 unique element names. The problems associated with maintaining or evaluating a system of this size are formidable.

## FORTRAN

Originally designed in 1954, Fortran [1,2] has continued to evolve during the past 32 years. Today Fortran is known for its simplicity, power, flexibility, and versatility. Fortran compilers are noted for the compact, fast and efficient code they produce. All these factors have contributed to Fortran's wide and varied acceptance within the programming community. Excepting perhaps only Cobol, Fortran is today the most widely used computer language. It has been used for tasks ranging from scientific applications to text processors, from compilers to natural language translation systems, and everything in between. How well the original designers wrought; they could not have begun to guess at the far-ranging success of their invention.

Fortran is today also known for the problems that attend its use. There is a wide and growing perception that large Fortran programs are inherently incomprehensible, unreliable, unstable, and unmaintainable. Over the years, much has been learned about programming language features that aid in the generation of maintainable and reliable code. Many of the lessons have come from experiences, good and bad, of the Fortran community. Much of the impetus behind the development of newer programming

languages, such as Pascal and ADA, is a reaction to both the real and perceived problems associated with Fortran.

## STATIC ANALYZERS

Program understandability and maintainability are tightly coupled. The more difficult a program is to understand, the more difficult it is to maintain. When faced with the problems of analyzing, evaluating, and maintaining a large program, more than will and skill are required. Some assistance is necessary if programmers are to maintain their sanity, much less accomplish the task. This is the role of automated tools.

One such tool is the static analyzer. As defined by Berns [3]:

> "A static analyzer is a program whose purpose is to expose, or provide information sufficient to enable the programmer to expose, programming discrepancies related to misuse or poor use of a programming language. 'Discrepancies' include errors, possible errors, poor usages (according to some general programming standard), suspicious usages, and clutter, but do not include algorithmic or data dependent errors."

A static analyzer is a software tool that examines a program's source code and reports on the correctness of syntax, semantics, element usage, and interface usage throughout and across all its constituent modules. It may also produce significant documentation of the program analyzed.

A traditional Fortran compiler processes a program module by module. That is, the compiler never sees the entire program at a single time. It has no knowledge of the other inter-communicating modules of a program when it compiles a module. The traditional compiler's function is to diagnose syntax and semantic discrepancies within a module, producing object code for correct source code.

Like the traditional compiler, a static analyzer processes each source module individually. However, it also processes all of them as a whole, constructing from all the subprograms the single entity that is the program. It examines each subprogram, checking for valid syntax, correct and consistent use of element names, subroutine and function references, and argument descriptions. It also analyzes, in a single action, the entire source code of the set of modules that constitutes a program. A subprogram examined in isolation may appear proper and correct, when in fact its interfaces are erroneous. A static analyzer is unique in that it can detect the interface discrepancies that modularization of a program makes possible.

As an additional service, a static analyzer can generate a set of software quality metrics that numerically rate aspects of each subprogram. These metrics include the complexity factor of the subprogram, relative skill level required to understand and modify the subprogram, and the relative risk associated with modifying the subprogram.

## MAT

The Maintainability Analysis Tool (MAT) is a documenting static analyzer for Fortran. Originally developed in 1983 to analyze Fortran 66, Fortran 77, and DEC Vax Fortran [4,5], today MAT is hosted on, and analyzes Fortran for, DEC, IBM, Cray, Prime, CDC, and Hewlett-Packard computers [6]. MAT was developed and is now marketed by G. M. Berns of SAIC. Like several hundred others within government, other commercial companies, and SAIC, my group acquired MAT to assist in the evaluation, validation, and maintenance of Fortran software.

A Fortran program consists of both static definitional statements and executable statements. The definitions establish the attributes and interrelationships of program elements. The difficulty of understanding a program is more than just a function of logical complexity. Program difficulty is a function of how the dynamic portion of the program manipulates and controls the static elements. The original tenets of MAT were that the difficulty of understanding and maintaining a Fortran program is the sum of the difficulties of understanding its constituent elements, and that the difficulties of understanding the elements can be quantified by the use of carefully selected weights and factors.

After the first few times MAT was used to evaluate operational code, it was obvious that MAT provided metrics useful in evaluating the maintainability of the software. More importantly, MAT also found significant numbers of errors and problems in the source code. It was clear that, only after errors in the software were identified and corrected, did it make sense to deal with the metrics.

## A BIG PROBLEM

In 1983 my group was tasked to evaluate the maintainability of a large software program. In this paper the program is referred to as "But It's Gigantic" (BIG). BIG contains approximately 50,000 specification statements and more than 258,000 lines of executable Fortran code, divided into 3600 subprograms. The code defines in excess of 38,000 unique named elements, including variables, arrays, entry points, parameters, and common blocks.

## THE FIRST LOOK AT BIG

BIG had been classified as operational software for a year before my group began the maintainability analysis of the program. The scope of the task was to assess the quality of each module of the program, including determinations of actual and possible source-level errors, but not to evaluate the correctness of module logic.

Before discussing the MAT analysis of BIG, one point must be made. We do not imply that the BIG system has unusually poor code. MAT has been applied to tens of thousands of production Fortran modules. It has found thousands of errors and problems, including major discrepancies, in other production Fortran systems. Rather, our view now of the BIG system is that it unfortunately appears to be the norm for production Fortran systems.

Logically, BIG is divided into four major subsystems (A, B, C, D) that interact heavily. MAT was employed to analyze each of the subsystems. The result of each analysis was the same: the BIG operational software had a large number of problems. MAT found numerous hard Fortran errors. Many modules were found to be excessively large, and intermodule communications were suspect.

The first MAT analysis results of Subsystem D were typical of all four of the subsystems. Of the 492 Fortran modules analyzed, source-level errors of various types and severities were found in 22 modules, with errors potentially existing in many more. Of the 22 modules, a total of 16 modules contained references to variable elements that were never explicitly given values. In each instance, the offending symbolic name was spontaneously and implicitly declared in the Assignment statement in which it first appeared.

Altogether, MAT found imperfections of various degrees in 317 of the 492 modules. That is, MAT diagnosed problems in 64% of the Subsystem D Fortran modules, with an average of 1.2 diagnostics per module and a maximum of six diagnostics per module.

## PATTERNS

Over the next few years, we used MAT to analyze multiple release of BIG. We continued to find more errors and problem areas. At last count we had documented over 275 hard Fortran errors, 500 interface mismatches, 110 global element discrepancies, 35 mixed-residence common blocks, and an uncountable number of poor program usages.

The problems clustered into two patterns. The first pattern was the relationship of unit testing to program size. The more lines of source code in a program (which roughly translates to more subprograms), the less comprehensive the unit testing of each subprogram appeared to be. The large number of errors that MAT found in the source code indicated to us that, at least in some cases, individual Fortran modules were not being fully reviewed and unit-tested.

Apparently the sheer number of modules in the BIG system caused the developers to combine the individual modules into groups of "testable" code as soon as possible. The prevailing reasoning may have been that the fewer integrated code groups would require fewer tests than full unit testing of all the individual modules. The developer seemingly believed that testing such code groups would be less costly and take less time to perform than full unit testing, and that their results would be as satisfactory. The MAT analysis showed that this idea is invalid, that a full spectrum of testing (unit, subsystem, and system) is necessary. A minimally acceptable set of unit tests should exercise every line of code and every branch at least once. Had this been done with these modules, all of the MAT detected errors and potential errors would have been found and presumably corrected.

The second pattern that emerged was that the same types of Fortran usage problems appeared in module after module. Six classes of discrepancies were frequently found: implicit data typing and spontaneous definitions, referenced but not set symbolic names, miscalling subprograms, inadequately controlled common blocks, syntax and semantics errors, and clutter.

## IMPLICIT DATA TYPING AND SPONTANEOUS DEFINITIONS

Our MAT analyses showed that Fortran's implicit data typing feature was clearly the most error-producing part of the language. Approximately 60% of the errors we found in the BIG system were due to this "feature". Implicit data typing allows the data type of an element to be declared based on the first letter of its name. It also allows the spontaneous definition of symbolic names: A local element may be defined and data typed by its first appearance in an executable statement.

Failure to enforce explicit data typing leads to errors of all kinds. We often found modules with spontaneously defined names due to improper key entry. Keying errors included transposed characters, wrong characters, missing characters, and extra characters. All generated legal, but unexpected symbolic names. In some instances we detected spontaneously declared elements caused by the extension of a line beyond column 72. We even found statements that were clearly "incorrect" but still

yielded legal Fortran! Consider the following typical cases from the BIG software:

```
IF (BRANCH .EQ. FALSE)...

ICTCOL = . . . . . + ICICOR
(Underlined C is in column 73)

IF (....) THEN I = 0
ELSE I = 10

WRITE (UNTINUM)...
WRITE (UNITNUM)....

SUBROUTINE INIT
COMMON /NEWND/ NAME, ISTAT,  . . .
          .
          .
SUBROUTINE GNAM
COMMON /NEWMD/ NAME, ISTAT,  . . .
```

These legal but incorrect examples result in the spontaneous but unintended definitions of symbolic names FALSE, ICI, THENI, ELSEI, UNTINUM, and NEWMD. Each looks so right that programmers did not detect the error.

## REFERENCED BUT NOT SET SYMBOLIC NAMES

Another frequently occurring problem found in BIG was the referencing of a symbolic name that was not given an explicit value. In many instances the problem was an outgrowth of implicit data typing. The Fortran 77 specification [7] states that undefined elements do not have a predictable value. As implemented, most compilers and linkers do provide a default value for symbolic names. However, the default values assigned range from -1 to 0 to the largest possible integer value. Some linkers "assign" whatever value happens to be present in the memory location at execution time. The DEC Vax compiler assigns a default value of zero to an uninitialized element. However, the predictability of such values over time is questionable. In the past, DEC has changed the internal representation of an element from one compiler to another. For example, the internal representation of Logical True/False changed between the RSX-11 Fortran compiler and the Vax Fortran compiler. There is no question that reliability, maintainability, and transportability suffer when symbolic names are referenced without being set.

## MISCALLING SUBPROGRAMS

In a subprogram call, the number of actual arguments must be the same as the number of dummy arguments, and the data type

(Real, Integer) and element type (variable, array) of an actual argument must match the data type and element type of the corresponding dummy argument of the called subprogram. All mismatches are errors [7] and can cause problems that are very difficult to trace. In the BIG system we found over 200 subprograms repeatedly referenced in one or more of these illegal ways. Tables showing each type of mismatch from just one of the BIG subsystems (Subsystem A, consisting of 1,401 subprograms) are combined in the following pages, extracted from a MAT report. For the sake of brevity, only the first few lines of each table are included here. In some instances the original tables each ran more than four pages. For our purpose the first lines of each table are representative of the errors detected.

Because of the implementation techniques employed by the DEC Vax, data and element mismatches may sometimes work correctly despite the errors. For example, I*2 -- I*4 mismatches in both directions work correctly on the Vax as long as only the low-order 15 bits of data are used. However, storing a four-byte result into the location of a two-byte element can destroy adjacent elements. Other computers may be less forgiving.

In numerous instances we found suspicious treatment surrounding constants passed as actual arguments to other modules. Examination of the called modules showed that they changed the value of the corresponding argument on some logic paths, but only referenced the argument on other paths. If the developer knows exactly how the logic works, if the input data are always exactly as expected, and if a maintenance programmer never changes the logic, then perhaps the program will work "correctly" for a while. However, this is always an extremely unwise practice and was vigorously decried in our reports.

## INADEQUATELY CONTROLLED COMMON BLOCKS

MAT detected every possible type of difference in the common blocks of the BIG system. Every action that could be taken to increase the difficulty of understanding and provide increased opportunity for error in the software was detected. MAT found problems in:

common block size
number of elements in the common block
name of the element
kind of element (i.e., variable or array)
data type of element (real, integer, complex)
number of array dimensions
size of array dimension
bounds of array dimension
equivalence position in the common block
common element name in multiple common blocks
common element name the same as local element name

TABLE 2-1 DATA TYPE MISMATCHES OF ARGUMENTS

| called subprogram | arg. nos. | required data types | actual data types | calling subprogram |
|---|---|---|---|---|
| ACHD | 5 | I*4 | I*2 | COVD, LATD |
| ADD | 2,3 | R*4 | R*8 | EDGZ, LOAD, SCN |
| BYPA | 1,2,3 | I*2 | I*4 | CURD, DISA, FLUS |
| CRTS | 2 | I*2 | I*4 | CREA, CREA |
| HGEO | 1,2 | R*8 | R*4 | APMK, CALC, DEGMN, DSAM, RTSY, WATW |
| HL | 1 | R*4 | R*8 | APX, MEWP, MIWP, MWPT, OFFS, SPAC |
| INF | 1,2 | I*2 | I*4 | CURP, PICK |

TABLE 2-2 ELEMENT TYPE MISMATCHES OF ARGUMENTS

| called subprogram | arg. no. | required elements | actual elements | calling subprogram |
|---|---|---|---|---|
| ADDL | 9 | array | var | AKNT, INSM, INSR, LOAD, MAK, OLVP, OLVP |
| DELL | 9 | array | var | EMPT, RIDS |
| DGRE | 1 | array | var | INFO, NOB, XMI |
| INFO | 2 | array | var | GETB, IDET, IHGH, ISEG, ISGS, IVIS, IVUP |
| INFOU | 2 | array | var | ALLO, CHARJ, CHARQ, CHARS, CLI, CLOS, CURP, FRAM, LINE, LLU, LMP, LOCS, MAR, SETR, TEM, TFUN |
| NOBT | 2 | array | var | INI, DELE, DISA, DIS, FLUS, FLUSH, GET, GETP, IC, INI, IPRM, RLOC, SETP, TERM |

TABLE 2-3 ARGUMENT NUMBER MISMATCHES

| called subprogram | required no. of arguments | actual no. of arguments | calling subprogram |
|---|---|---|---|
| ADD | 3 | 4 | CPAD |
| DELL | 9 | 8 | CAMI, DELI, DELT, PAM |
| DRAW | 8 | 10,7 | TGTI |
| ERRM | 0 | 1 | DENW, DSTW |
| ERR | 8 | 9 | TEC |
| ERRS | 5 | 1 | REST |
| MAPD | 1 | 0 | TERM |
| NDXS | 0 | 2 | CAM |
| RATE | 2 | 3 | CMP, CPA |

TABLE 2-4 INCONSISTENT FUNCTION DATA TYPES

| called subprogram | required data type | actual data type | calling subprogram |
|---|---|---|---|
| MPSTA | I*4 | I*2 | MAPD,MSET,OBJEC |
| SYS$BINTI | I*4 | R*4 | KICK |
| SYS$WAITFR | I*4 | R*4 | CINT, DSDE, OLS |
| SYS$CLREF | I*4 | R*4 | CINT, DSDE, KICK, OLS, SCND |
| SYS$SETEF | I*4 | R*4 | AS, CINT, DSDE, OLS, OPD, SCEN |
| SYS$WAITFR | I*4 | R*4 | CINT, DSDE, OLS |

In many instances a common block was both hard-coded into a subprogram and appeared in an "include file" referenced by other programs. In approximately 5% of such cases, the definition of the hard-coded common block differed from that of the "include file" common block. Such a situation can not only cause subtle execution errors, but it can also be a severe challenge to configuration management. One purpose of the "include file" is to centralize commonly used elements. Should the elements of a common block require modification, the configuration manager need only edit the "include file" and recompile all subprograms referencing the "include file". Configuration managers who think a common block is only defined within an "include file" and vigorously control that common block can be the unwitting agents of serious errors if hard-coded versions of the common block also exist.

All instances of a common block definition should be identical in every respect. To program in any other way leads only to confusion and errors. Programmers rely on consistency among modules. The symbolic name ABC defined as the first element of labeled common /XXX/ in module S is nominally expected to be the same ABC when it appears in module T in labeled common /XXX/. Yet, in BIG subprograms, ABC appeared both as a common block element and a local variable, and appeared in different locations of the same common block in different modules.

## SYNTAX AND SEMANTICS ERRORS

Often the BIG programmers seemed to have been confused about what constitutes a legal Fortran statement. Consider the following example discovered by MAT:

```
IF (IBRACK .EQ. 0) THEN IBRACK = 1
```

The programmers obviously thought they were employing the Fortran IF-THEN-ELSE structure. In reality, the code implicitly defined a new variable (THENIBRACK), to which it then assigned a value.

A second classic Fortran error, also resulting from the lack of strong data typing in Fortran, appeared in the following code:

```
OUTLINESFILE  =  3
REWIND COVERAGE OUTLINES FILE
REWIND (UNIT = OUTLINESFILE)
```

The programmer apparently intended the second statement to be a comment statement. However, the "C" in column 1 is missing. This results in the Fortran compiler generating statements to rewind units COVERAGEOUTLINESFILE and OUTLINESFILE. The code rewinds I/O units three and zero -- probably not what the programmer intended! Unit zero is a valid unit and may actually

be in use when it is rewound. Note that strong data typing and adequate unit testing would have prevented this problem from reaching operational software.

CLUTTER

On the surface, code "clutter" is easy to describe. Clutter is the term for program elements whose removal does not alter the operation of a module. The question is, can clutter always be identified accurately? The majority of operational Fortran code that we have examined apparently contained substantial amounts of clutter. Clutter obscures the logic of a module. It adds noise to the code and makes understanding of the module harder, increasing the difficulty of maintenance. However, the most critical aspect of clutter is the subtle relationship it bears to discrepancies.

We found that more than 90% of the BIG system modules contained some form of clutter. The clutter found in the BIG system included unused local symbolic names, unused common block symbolic names, wholly unused common blocks, unused dummy arguments, unused entry points, and wholly unused modules. Over 20% of the modules contained wholly unused common blocks. In some cases, modules contained more than 30 unused common blocks.

It is difficult for maintenance personnel and analyzers to separate clutter from an error. Because Fortran does not require strong data typing, what at first glance appears to be noise may instead be a discrepancy. Consider the following sample subroutine:

```
SUBROUTINE X (DEF, MAX)
GHI = 1
MMAX = 4
DEF = 2 + MMAX
RETURN
END
```

The argument MAX is never used, but the local element MMAX is assigned a value and later referenced. The local element GHI is set, but never referenced. It is possible that the argument MAX and the local element GHI are clutter and can be removed. However, things may not be as simple as they appear. Certainly the possibility exists that MAX is a mistyping of MMAX (or vice versa), and that the symbolic name GHI appears in a common block required by, but not present in, this module. In this case, the problem is not the "clutter" of GHI, but the real error of a missing common block. There may be clutter, errors, or both in this module.

The presence of unused common blocks, of which BIG had an inordinate number, can prove to be an error just waiting to

happen. The effects of such an error are not constrained to be manifested solely within the module actually in error. Instead, the effects of an error associated with a common block can be widespread, propagating throughout the system. By design, a common block is a "window" offering access both into and out of a subprogram. A mistyped name may happen to be identical to one that actually is defined. The more defined names in a module, the greater this possibility. While a collision between names is always possible, elements that are not clutter must be present. By definition, this is not the case with clutter. Its presence raises the probability that this kind of error will occur.

## "KILL THE MESSENGER" SYNDROME

The nominal goal of a software developer is to produce a working and functionally useful program that satisfies the specifications of the customer, within the constraints of time and budget. Professional pride dictates reasonable care in the design, development and testing of the software. Realizing that customers often do not initially know what they really want or need, the developer should still strive to produce flexible software and be sensitive to any problems in the operation of the system.

The role of independent validation and verification (IV&V) is adversarial by nature. In evaluating the BIG system, our job was to detect problems in the system and assess the maintainability of the software. We wished to have our efforts respected, even though we knew we would be treated with some suspicion by the developer. In our initially unenlightened way, we expected that people would want to know about software problems and be prepared to correct them, rather than remain ignorant of the problems. Software problems will surface, often at the most inconvenient time.

The results of the MAT analyses of BIG were a series of reports that documented the software errors, possible errors, and poor practices found in the source code. Each report ended with a conclusion about the state of the software and a set of recommendations suggesting ways to remedy the problems.

Initial reactions to the first reports were interesting, to say the least! As discussed previously, the majority of errors detected in the code were a direct result of the lack of strong data typing in Fortran. Scores of hard errors found by MAT were due to the implicit declaration of symbolic names. Yet, DEC Vax Fortran offers the IMPLICIT NONE compiler directive that directs the compiler to mark as an error any symbolic name that is not explicitly declared within a subprogram. In almost every instance, use of the IMPLICIT NONE statement would have signaled the problem the first time the subprogram was compiled. Yet, in release after release of the software, we continued to document

dozens of errors resulting from the implicit declaration of elements. Often a module contained errors reported in a previous release, as well as new errors of the same kind introduced in the latest release.

Could we convince the developer to require the use of IMPLICIT NONE? No! The developer responded that use of IMPLICIT NONE, and the attendant requirement to declare explicitly each symbolic name, placed too great a burden on the programmer. "The programmer might as well be programming in Cobol!" Despite all the community has learned in the past 32 years and the hard errors we could document, explicit definition of elements was "unnecessary, not worth the effort, and besides the program seems to be working O.K." So the errors remained, because they did not seem to be causing any problems, or the errors were laboriously removed one by one, and with each new release of the software new errors of the same type appeared.

A second example of how the MAT reports were initially received showed the other extreme. The MAT report for one release of BIG documented an IF-THEN-ELSE structure that performed a test on an element that was never explicitly given a value. The logic was such that the ELSE clause could never be executed. The problem was reported to both the customer and the developer. A few months later we began hearing complaints that MAT had caused a serious problem with the BIG system. Discussions were held to decide if MAT should ever be run against the software again.

After a good bit of investigation, we were able to reconstruct the entire story. A programmer had read the MAT report and, while making other modifications to the module (call it module P), decided to fix the reported error by adding code that established the correct value for the transgressing element. The changed module P compiled without error and was included in the next release of BIG. However, when BIG was delivered and testing began, the system crashed almost at once. The "fix" did not work. Since the "fix" was correct, the problem had to be with MAT. The attitude was, "If we fix it and the problem is worse than before, it must be MAT's fault".

The reality was a bit different. On later and more careful examination, the programmer determined that the offending module P called modules Q and R. Earlier, the programmers of modules Q and R had concluded that the output of module P was not what was expected. Instead of correcting module P, they changed the interface to modules Q and R to compensate for the problem. The interface changes were never documented. Now along came the programmer who fixed module P according to our MAT report, not knowing about the implications of the fix. The programmer failed to test the fix in a complete manner. Modules Q and R were now inconsistent with the corrected logic of module P. So, the first time the corrected logic path was executed....

Errors in software cause unexpected results in the execution of the program and sometimes in the mind of the programmer. Fixing a reported error does not guarantee that the system will run better; simply to fix an error without carefully evaluating the cause and effect of the error can invite disaster.

A strange reverse logic appeared here. Module P was reported by MAT to contain an error. When the error was corrected, the system worked worse than before. Because the system was "working" before MAT reported the error, the problem must be MAT. If MAT creates more work for the developer, the best solution is not to use MAT. If you do not know about errors, there are no problems. Reporting problems causes problems; thus "Kill the messenger"!

Cases like this provided us with interesting lessons in human behavior. In the case of spontaneous element definition, the multitude of errors were, for the most part, treated as unimportant because the software seemed to work. It was not deemed necessary to correct a hard error that caused no observable or identifiable problem in system execution. The idea that the effects of the error might be subtle or masked by subsequent processing, or that the specific logic path containing the error had yet to be executed, had little credence. This was true despite repeated instances where problems were eventually traced back to earlier MAT reported errors.

## PERSISTENCE PAYS OFF

This story ends on a high note. It has taken three years, but attitudes have changed. Over this period we accumulated a substantial mass of MAT documented evidence showing serious flaws in the programming practices employed in BIG. At some point the sheer weight of evidence changed the way people viewed the information produced by MAT's static analysis. When we started we were outside the fence protecting the program, but sometime along the way the fence was moved. MAT and its use became institutionalized, resulting in the circle of MAT users expanding throughout the project.

Confirmation of the changed attitude is shown in numerous incidences. In one instance, the day following release of a MAT report documenting a host of hard Fortran errors not previously identified, an unscheduled tape containing corrections for 20 of the errors arrived. Subsequent to that, when an outside agency came to evaluate the project, the use of MAT was held up as one example of sound project management. In a third instance, MAT analyses of two contending software packages were employed to assist in the source selection process.

Problems reported by MAT are now routinely entered into the formal BIG error reporting system and carefully tracked until corrected. Some programmers are using the information from MAT to produce better software during code development. As a first step toward reducing the potential for error, there is now an effort underway to explicitly declare all common block elements and put them in "include files". More importantly, people's attitudes about error reporting have changed. Now they accept the idea of MAT's identifying errors in BIG even though the errors may not yet have been encountered, and they are willing to commit the resources needed to fix the problems. Each new release of BIG is now completely processed by MAT before it is installed in the field.

## SOME UNEXPECTED USES FOR A STATIC ANALYZER

Originally envisioned as an aid in the evaluation of software maintainability, a static analyzer such as MAT has the potential for many uses. Not only is MAT useful after the fact, to assess software maintainability, but it is also a powerful tool in the development and debugging of software. MAT reports errors and problems normally not detected by compilers, such as unset elements, common block inconsistencies, and calling sequence discrepancies. A subprogram for which MAT reports no error, poor practice, or warning messages is less likely to contain entire classes of errors. A static analyzer cannot detect data dependent or logic errors, but experience has shown that it can detect many common programming mistakes.

Because MAT has access to every symbolic name in a program, it can produce a complete symbolic name cross-reference table for the entire program. Most compilers can only cross-reference symbolic names within a subprogram. For each symbolic name, MAT provides a complete description, including (where appropriate) data type, size, use by subprogram, and whether the element is set, referenced, or used as an argument. This documentation is valuable for program development and maintenance.

Recently we were tasked to transport a 30,000 statement program from the DEC Vax to another vendor's hardware. The program was divided into 474 subprograms, contained 8,150 symbolic names, and employed the full range of Vax Fortran extensions. The target system supports a more standard Fortran 77 and is less forgiving than DEC Vax Fortran. Because MAT marks the use of many Vax Fortran extensions and provides a complete cross-reference listing, many of the transportation problems we faced were manageable. During a two month period, the code was successfully installed and executed on the target system. Without MAT or a similiar tool, the job might not have been performable with the available resources.

## MAT-INSPIRED RECOMMENDATIONS FOR IMPROVING SOFTWARE QUALITY

Aside from correcting the errors identified by MAT, there are a number of ways that the quality (e.g.; reliability and maintainability) of a Fortran program can be markedly improved.

1. Define explicitly every symbolic name in the software. Do not rely on any default definitions.

2. Add to each module the IMPLICIT NONE statement, if it is part of the Fortran dialect used.

3. Ensure that all subroutine call sequences and function references are consistent in number, type, and kind with the called module's argument list.

4. Require a single definition of a common block resident on an "include file", if "include files" are supported. Every module that requires the common block should reference the "include file".

5. Remove code that is clutter, especially wholly unused common blocks.

These five recommendations can considerably increase the reliability and maintainability of Fortran software.

## CONCLUSION

Evaluation and maintenance of large software programs present a set of unique problems and challenges. One aid useful in the struggle is the static analyzer. Although a static analyzer cannot handle issues of data dependencies or algorithmic correctness, it can detect a large number of severe usage problems.

With the MAT static analyzer, we were able to identify and document hundreds of problems in the BIG system. However, as useful as it is, a static analyzer is only a tool. The data it provides must be carefully and completely analyzed and understood by a person. Someone with skill and experience must ultimately examine the source code to verify and perhaps describe the exact nature of problems. Used appropriately, a static analyzer can distill the myriad elements of the code into an organized and manageable form.

It has taken three years and numerous reports, but our work with MAT has resulted in better software in BIG. We have made many recommendations, based on what has been uncovered in the source code itself, for improving the quality and maintainability of the BIG system. Many of these recommendations have found their way into the the software. Equally important, our ability

to consistently and rapidly detect whole classes of errors has helped change the attitude of people toward error detection and correction. The best that can ever be said of any software is that all known errors have been corrected and that all known problem areas have been eliminated. With MAT we have been able to discover and document large numbers of errors and problem areas. In time, we may truthfully be able to say that all known errors and problem areas have been removed from the BIG system.

BIBLIOGRAPHY

1. Fortran II for the IBM 704 Data Processing System. Reference Manual C28-6000. International Business Machines Corporation, New York, 1958.

2. Backus, J. W. et al. The Fortran Automatic Coding System, 1957.

3. Berns, G. M. MAT, A Static Analyzer of Fortran Programs, and the Most Common Fortran Reliability Problems, Proceedings of the Digital Equipment Computer Users Society, December, 1984.

4. Berns, G. M. New Life for Fortran, Datamation, 1 September 1984.

5. Berns, G. M. MAT, A Program That Analyzes Fortran Programs. Science Applications International Corporation, Arlington, VA, June 1984.

6. Berns, G. M. User's Guide for MAT Version 10. Science Applications International Corporation, Arlington, VA, June 1985.

7. American National Standard Programming Language Fortran, X3.9 - 1978. American National Standards Institute, New York, 1978.

# Session 3

## TEST TOOLS

*"UDT: A Tool for Debugging and Testing Software Units"*
Bruno Alabiso, Motasim Najeeb, Craig Thomas, Intel Corporation

*"TCL and TCI: A Powerful Language and an Interpreter for Writing and Executing Black Box Tests"*
Arun Jagota, Vijay Rao, Intel Corporation

*"T: The Automatic Test Case Data Generator"*
Robert M. Poston, PEI (Programming Environments, Inc.)

# UDT: a Tool for Debugging and Testing Software Units

*Bruno Alabiso*
*Motasim Najeeb*
*Craig Thomas*

Intel Corporation
Development Systems Operation

## ABSTRACT

This paper describes a product that combines the functions of a software debugger with those of a testing tool (UDT, Unit Debugging and Testing). Debugging and testing are both processes aimed at enforcing consistency between the *expected* behavior and the *actual* behavior of a software system. UDT employs a formal language to define the *expected* behavior of a system and/or its subparts: the *expected* behavior is compared at run time with the *actual* behavior. UDT operates at *unit* level, in the sense that it supports the debugging and testing of single, isolated subparts of a system. UDT incorporates the functions of an *automatic test generator*. Finally UDT employs an advanced *window oriented human interface* to present multiple views of the state of a system to the user.

## BIOGRAPHIES

### Bruno Alabiso

Bruno Alabiso received an MS degree in Computer Science from the University of Naples, Italy (1975). He has worked as Principal Research Engineer in ITT, England. He has operated in the areas Strategic Planning, Product Architecture definition and Project Leadership in Intel Corp. Areas of professional interest: Microprocessor Applications, Real Time Operating Systems, and Software Development Tools.

### Motasim Najeeb

Motasim Najeeb graduated from the University of Southwestern Louisiana, Lafayette, Louisiana, with a MS degree in Computer Science. He is presently working for Intel Corporation as a Software Engineer. His areas of interest include: Software Development Environments/Tools and Advanced Human Interfaces.

### Craig Thomas

Craig Thomas is a Software Development Engineer at Intel's Development Systems Operation. After receiving his computer science degree from Washington State University in 1984, Craig joined Intel. He has concentrated his focus on the development of debug and performance analysis tools.

# UDT: a Tool for Debugging and Testing Software Units

*Bruno Alabiso*
*Motasim Najeeb*
*Craig Thomas*

Intel Corporation
Development Systems Operation

## ABSTRACT

This paper describes a product that combines the functions of a software debugger with those of a testing tool (UDT, Unit Debugging and Testing). Debugging and testing are both processes aimed at enforcing consistency between the *expected* behavior and the *actual* behavior of a software system. UDT employs a formal language to define the *expected* behavior of a system and/or its subparts: the *expected* behavior is compared at run time with the *actual* behavior. UDT operates at *unit* level, in the sense that it supports the debugging and testing of single, isolated subparts of a system. UDT incorporates the functions of an *automatic test generator*. Finally UDT employs an advanced *window oriented human interface* to present multiple views of the state of a system to the user.

### 1. Background

Debugging and testing play a crucial role in the development and maintenance phases of software projects. Both activities can be viewed as efforts aimed at ensuring consistency between the *expected* behavior of an application system and its *actual* behavior.

Debugging is generally well supported by tools (*debuggers*) whose main purpose is to instrument the application system in order to facilitate the identification of faulty code, i.e. of code that produces unexpected behavior. 'Instrumenting' an application system normally consists of providing functions to start and stop execution (such as breakpoints and stepping) and others to inspect or modify the state of the system.

Testing is in general poorly supported. In the typical development environment, test suites are written by hand. Often test results are analyzed by hand as well. Testing procedures vary enormously from one development environment to another, with different degrees of formalism being enforced.

Most important, the *expected* behavior of an application system is normally expressed in a nonformal way: specifications are written in plain English, if they are written at all. Under these circumstances, it is impractical to provide computer tools that automatically check whether the system is behaving as specified.

What commonly happens is that the programmer works with informal specifications and the product is delivered for evaluation/testing as soon as a *subjective* satisfaction with its behavior is achieved.

Although several attempts have been made to introduce formal languages for product specification [2], very little attention has been paid to the integration of these specifications into the debugging and testing phases.

Another common problem in testing is that *software products are tested only when they have been*

*entirely constructed*; in other words, testing is not performed on individual components but only on the overall system. This method has two negative consequences:

(i) A system built from untested parts is much more likely to contain faults than a system whose parts have been individually tested.
(ii) Once an error has been detected in a system whose parts are untested, it is generally difficult to pinpoint the part that caused the malfunction.

UDT (Unit Debugging and Testing tool) attempts to solve the previously discussed problems by:

(i) Allowing parts of a software system to be debugged or tested in isolation
(ii) Providing a specification language (very similar to the programming language) to define the expected behavior of individual system components
(iii) Checking specifications against the actual behavior of the system at run time
(iv) Generating test suites automatically from the system's specifications.

In addition, UDT is a high level symbolic debugger incorporating conventional debugging functions (such as execution control, memory inspection, and memory modification) with a multi-window human interface paradigm.

## 2. UDT Approach

A typical nontrivial software system is comprised of many subsystems that are generally developed in parallel by various software engineers in the development team. The engineers are aware only of the interfaces of the different subsystems that will eventually be integrated to form the complete software system.

A given subsystem might fall anywhere in the hierarchical tree representing the entire system, and hence may consist of:

(i) Subprograms that are called by subprograms included in some other subsystem being developed in parallel, or by subprograms still unimplemented (bottom-up approach)
(ii) subprograms that make calls to subprograms belonging to other subsystems, or to subprograms that are still unimplemented (top-down approach)

Testing of isolated subsystems during development is generally unsupported, and manual methods are employed in either of these cases to test such units of code. Testing of subprograms in the first case is termed *unit testing*. For unit testing, a test driver is created to mimic the calling subprogram. The subprogram is then repeatedly called from the test driver with a set of manually created test data to verify the correct behavior of the subprogram. Testing of subprograms in the second case is known as *stubbing*. In this case, stubs are created for subprograms that are still undeveloped. A third type of testing that is useful during development is the assurance of correct *interfacing* between two subsystems. The interface between the subsystems is checked against a given set of specifications. This type of testing is termed *assertion checking*.

In general, typical software development uses a combination of top-down and bottom-up approaches; therefore, both unit testing and stubbing are common practices during the development process. Assertion checking, however, is uncommon because of a lack of testing tools. The test drivers and stubs result in throw-away code and are discarded after system integration. Moreover stubs mainly serve the purpose of satisfying unresolved externals required for successful compilation of the subsystem for execution. They do not contribute in the testing of the calling subprogram. Test case generation and verification is usually manual, and hence a time consuming process.

UDT provides mechanisms to automate the functions of stubbing, unit testing, and assertion

checking through a formal specification language. The emphasis of UDT is to allow early testing of both aggregate software systems and independent software modules in isolation.

### 2.1. Formal Unit Specifications

UDT provides a specification language that is very similar to the language in which the application is implemented. Since UDT provides testing and debugging of applications implemented in several languages (C, Pascal, PL/M, ADA), a specification language is provided that corresponds to each language supported. For the purpose of this document, the application under test is assumed to be in C; hence, examples are given in the specification language similar to C.

During any stage of the implementation cycle, the user can test the actual behavior of the application against its specifications by creating a set of information entities called Unit Specification Tables (USTs). A single UST serves the multiple purposes of stubbing, unit testing, and assertion checking; separate *UST compilers* are used to generate the code sequences that specify the behavior for each testing feature. Each entry in the UST is known as a *unit specification.*

Each subprogram under test may be associated with one or more USTs. For stubbing, the UST provides behavior for subprograms that are unimplemented. For unit testing, the UST provides a driver for subprograms that do not have an entry point. This driver repeatedly calls the subprograms with appropriate values for the input and then checks the output values against a given set of specifications. For assertion checking, the UST checks subprogram calls *at the interface*, by comparing input/output vectors against the specified behavior of those subprograms. Figures 1, 2, and 3 illustrate these three cases.

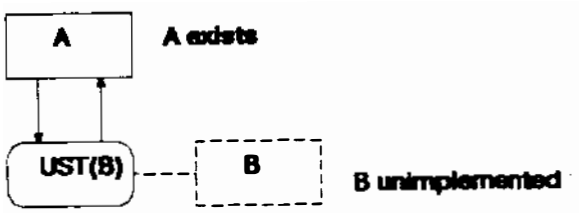



**Figure 1. STUBBING**

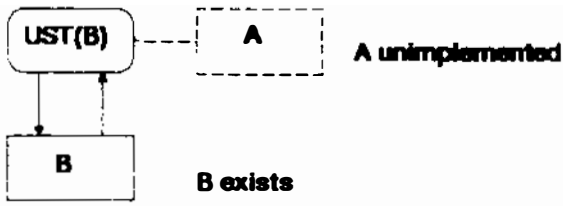



**Figure 2. UNIT TESTING**

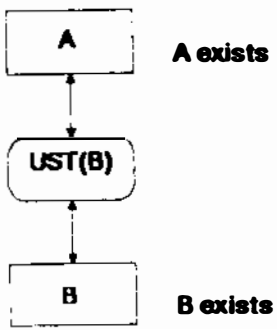



**Figure 3. ASSERTION CHECKING**

A Unit Specification Table is in essence a modified decision table [6] [9]. However, rather than determining a set of executable actions corresponding to a given set of conditions (as in the case of a decision table), a set of exit conditions are established for each corresponding set of conditions. The format of a UST is shown in Figure 4. Specifying the behavior of a desired subsystem requires various identification fields in the UST, allowing UDT to execute one of the three testing schemes: stubbing, assertion checking, or unit testing.

| SUBPROGRAM: my_proc | ASSERTION: ENABLED | | STUBBING: DISABLED | |
|---|---|---|---|---|
| # | ENTRY CONDITION | EXIT CONDITION | STUB | ON ERROR |
| 1 | a.b > 0 && Rec_Valid(a) | a.c = = a.d + a.e | a.c = a.d + a.e | |
| 2 | a.b <= 0 && Rec_Valid(a) | a.c = = a.d - a.e | a.c = a.d - a.e | |
| 3 | !Rec_Valid(a) | strcmp(a.str,"INV") = = 1 | strcpy(a.str,"INV") | |
| INVARIANTS: num_records | | | | |
| INPUTS: struct my_rec a | | | | |
| OUTPUTS : struct my_rec a | | | | |
| ON ERROR UPON ENTRY: | | | | |
| ON INVARIANT ERROR: | | | | |
| DEBUG PROCEDURES SECTION:<br>int strcmp(str1,str2)<br>char *str1, *str2;<br>{<br>...<br>} | | | | |

**Figure 4. UNIT SPECIFICATION TABLE**

The SUBPROGRAM field specifies the name of the subprogram whose behavior is described by the UST. The ASSERTION and STUBBING fields are toggle switches that can be enabled or disabled by the user. The ENTRY CONDITION and EXIT CONDITION fields contain boolean expressions that specify the expected behavior of the subprogram under test. ENTRY conditions and EXIT conditions are similar to the preconditions and postconditions defined by Dijkstra [1], Gries [3], and Hoare [4]. The STUB field contains statements, or the name of a procedure that describes the behavior of the subprogram being stubbed.

If the specified subprogram is being stubbed, the STUBBING field is enabled and the ASSERTION field is disabled. The ENTRY CONDITION field specifies the input conditions for the subprogram being stubbed. The corresponding STUB field contains statements that describe the behavior of the stub for those input conditions.

Conversely, if the specified subprogram is being assertion checked, the ASSERTION field is enabled and STUBBING is disabled. For assertion checking, both the entry condition and the exit condition of each unit specification must evaluate to TRUE. If the exit condition of a unit specification evaluates to FALSE when the corresponding entry condition is TRUE, the subprogram under test is not behaving as specified. The ON ERROR field lists the name of a debug function for a unit specification that fails when a subprogram is assertion checked. The debug function describes the actions to be performed by UDT if the corresponding unit specification fails during assertion checking.

The INVARIANT field lists all those symbols whose values remain unchanged when the specified subprogram exits. The ON INVARIANT ERROR field specifies the debug function to be executed when any values in the INVARIANT field change upon execution of the specified subprogram. This feature provides an effective way to detect undesirable side effects in a subprogram. The INPUTS and OUTPUTS fields list all the symbols of the specified subprogram whose values are used or

affected by its execution. These symbols are used for the automatic generation of test cases for the specified subprogram.

The ON ERROR UPON ENTRY field specifies the name of the debug function to be executed when *all* entry conditions in the UST evaluate to FALSE. In this case, either the application is not behaving as expected (i.e., the execution of the application caused an undesirable side effect since none of the expected states occurred) or a valid state exists in the application, but this state was overlooked by the writer of the UST. Finally the DEBUG FUNCTION SECTION lists all the debug functions that are referenced in the UST, and are local to it. UDT also provides for global debug functions that can be referred to by different USTs. The global debug functions reside in a library of debug functions.

### 2.2. Stubbing

If the application program contains calls to a subprogram that has not yet been coded, the behavior of the subprogram can be simulated by preparing a corresponding UST with the STUBBING field enabled. For simplicity, it is assumed in the following rules that the ASSERTION field is currently disabled.

In the case of stubbing, whenever the unimplemented subprogram is called, the following events occur in order.

(i) All entry conditions that evaluate to TRUE are marked (*Valid Entry Conditions*). If all entry conditions evaluate to FALSE, the ON ERROR UPON ENTRY debug procedure is executed.
(ii) At random, one *Valid Entry Condition* is selected and the corresponding STUB debug procedure is executed.
(iii) Control is passed back to the caller.

### 2.3. Assertion Checking

Any application system subprogram associated with a UST can be assertion checked by UDT as long as the ASSERTION toggle in the UST is enabled. For simplicity, the following rules assume that the STUBBING field is disabled.

In the case of assertion checking, when a call is made to a subprogram that has a corresponding UST, the following events take place:

(i) The values of all INVARIANT symbols are stored in a temporary buffer.
(ii) All entry conditions are checked. If they all evaluate to FALSE, the ON ERROR UPON ENTRY debug procedure is invoked. Otherwise, all entry conditions that evaluate to TRUE are flagged (*Active Entry Conditions*).
(iii) The subprogram is regularly called and executed.
(iv) On return, all exit conditions corresponding to the *Active Entry Conditions* are evaluated. If one or more evaluate to FALSE, an *Exit Condition Error* is generated. If corresponding ERROR entries exist in the UST, the specified debug procedures are executed. Otherwise, the application program is halted and a message is produced stating: the name of the subprogram that caused the assertion failure, the *Active Entry Conditions* at the time of invocation, and the corresponding exit conditions that evaluated to FALSE.
(v) The INVARIANT symbol values are compared with those stored on entry. If any of these values has changed, the ON INVARIANT ERROR debug procedure is invoked. If this procedure is not specified, the application program is halted and an INVARIANT ERROR

message is produced. This message identifies the symbols that failed the invariant check.

(vi) If the application program has not been halted as a result of an error (as described in the previous rules), control is passed back to the caller.

In case ASSERTION and STUBBING are both enabled, the process described above is performed, with the only difference that the subprogram is not called (refer to step [iii]), and instead, a stubbed debug procedure is executed (according to step [ii] in section 2.2).

### 2.4. Unit Testing

Under large development projects, modular units of code can be thoroughly tested at any stage of development without the need for test drivers. Testing is supported in UDT by two unique features:

*Automatic Test Generation*: the capacity to generate test suites semi-automatically from UST's, supported by UDT's *Test Generator*

*Test Execution*: the capacity to submit test suites, collect test results automatically, and compare new results with previous results, supported by UDT's *Test Executor*

The following two subsections discuss these two functions in detail.

#### 2.4.1. Automatic Test Generation

The ability to express the behavioral specifications of a system part in the UST promotes the automatic derivation of test vectors from the UST itself.

The entry condition fields of the UST define the *expected state* of the system when the corresponding part (subprogram) is invoked. It is therefore possible to construct an algorithm that generates pseudo-random values for the subprogram inputs that would cause one or more of the entry conditions to be evaluated to true. In this way the algorithm generating the test cases may exhaustively exercise all entry conditions defined in the UST.

The INPUTS and OUTPUTS fields in the UST are used by the test generation algorithm to identify and allocate memory for the test input and output variables.

A simple example will clarify this use.

Consider a subprogram *my_proc* that features the following INPUTS and OUTPUTS defined in the corresponding UST:

```
INPUTS: int a, int b;
OUTPUTS: int c;
```

Assume that entry condition #1 of the UST for my_proc is:

```
a == 1 && b < 10;
```

Test vectors that would satisfy this entry condition may easily be generated with values. For example:

```
a = 1; b = 3;
a = 1; b = 7;
```

If more than one entry condition is specified in a given UST, the algorithm for the generation of the

test vectors randomly selects different entry conditions to optimize test coverage. Before submitting the test vectors and invoking the subprogram *my_proc*, clearly memory needs to be allocated for *a*, *b* and *c*.

The choice of test values that satisfy given entry conditions is not always trivial. The example shown above considers a simple entry condition where all operands are variables. Entry conditions are not always this simple, since the syntax for UST entry conditions allows *function names* to be operands. In this case the test generator virtually needs to know the semantics of the function, so that it can select the correct actual parameters that force the function to return a value that satisfies the entry condition. A clarification is provided by an example.

Suppose that an entry condition specifies:

```
strcmp(x, y) == 0;
```

In this case, *x* and *y* are string pointers. The test generator should force the two strings to be identical.

Clearly, the test generator cannot perform this task without knowing what the function *strcmp* does. This problem is solved in UDT by allowing the user to define a *Test Generator Procedure* (or *TGP*) that accompanies the function used in the entry condition (called *Entry Condition Function*, or *ECF*). The TGP takes as input parameters the value that the corresponding Entry Condition Function (ECF) should return plus the values of the ECF parameters that have already been identified, are known or predetermined. For instance a constant actual parameter has a known value. The TGP contains a program fragment that chooses the undetermined input parameters for the associated ECF, which forces the ECF to return the desired value.

More precisely, suppose that the ECF has the form:

```
F(x1, x2, ..., xn)
```

Then the TGP would have the form:

```
TGP(ret_val, &x1, f1, &x2, f2, ..., &xn, fn)
```

where:

- *ret_val* is the value that the ECF should return.
- *xn* is the value of the nth ECF parameter chosen by the TGP.
- *fn* is a flag which indicates to the TGP whether the *n*th parameter has already been determined (for instance, it is a constant).

The correspondence between the TGP and its associated ECF is enforced by naming the TGP with the same name as the ECF with the addition of the postfix *_tgp*. In our example the TGP *strcmp_tgp* would have the form:

```
strcmp_tgp(ret_val, &x, f1, &y, f2)
```

It would be called by the test generator with the values:

```
strcmp_tgp(0, &x, 0, &y, 0)
```

(where the 0s after &x and &y mean *not selected*)

The TGP *strcmp_tgp* would force the strings pointed to by *x* and *y* to be identical.

A set of predefined, frequently used ECFs and corresponding TGPs will be provided in the *Test Generation Function Library*. The user has the ability to enrich this library with newly defined TGPs.

Alternatively, the user may choose to write a *Global Test Generation Procedure* for the *whole* UST: in this case the *Global Test Generation Procedure* will be invoked by the Test Generator to construct test vectors for the subprogram with which the UST is associated.

### 2.4.2. Test Execution

The output of the *Test Generator* for a given UST is a *test suite*. A test suite is a collection of tests. A *test* is a set of values associated with the INPUTS of the UST.

Naturally UDT provides facilities to hand-write test suites. The language for writing test suites is not specified yet.

A test suite can be submitted for execution in two different modes:

*Raw Testing Mode.* In this mode all tests belonging to a given test suite are submitted in a sequence for execution. The execution of a test consists of:

(i) Calling the subprogram associated with the UST with the input values defined in the test
(ii) Collecting the actual OUTPUT values in the test result buffer. Each test result buffer is associated with the corresponding test suite that generated it.
(iii) If the assertion checking mechanism is enabled for the UST, tests that result in entry condition, exit condition, or invariant failures are individually marked in the test result buffer.

*Regression Testing Mode.* In this mode, the tests belonging to a given test suite are submitted together with the associated test result buffer. Discrepancy of test results between the previous and the current run are individually marked in a new test result buffer.

UDT also provides facilities to merge test suites and test result buffers, as long as they are associated with the same UST.

## 2.5. Debugging Functions

The debugging features in UDT are symbolic in nature and are comparable to other symbolic source debuggers [5] [7] [8] [10]. A few enhancements, however, make UDT's features more attractive. These enhancements are discussed in the following paragraphs.

*The ability to load subunits of code.* The user does not need to build the entire application before debugging can start; UDT has the capability to load sections of code that have calls to unresolved routines or to load routines that have no entry point (i.e. a routine that has no entry code to initialize its environment). This capability allows the coder to debug a specific sequence of code without waiting for other sequences to be developed and linked.

*Breakpoints.* The user has three modes of breakpoints from which to choose: hard breaks, soft breaks, and condition breaks. Hard breaks are user set breakpoints set that remain in the program under debug until the user explicitly removes them. Soft breaks are breakpoints that are automatically removed when that breakpoint is reached during program execution. Condition breaks are set interactively in the debugger environment and are removed automatically when the break occurs. Program execution is halted by a conditional break when a specified boolean condition is met. For instance, the user could specify a conditional break with the following syntax:

**cbrk at 42 when alpha > 150**

This statement means that a program execution will stop at line number 42 when the variable *alpha* is greater than 150.†

*High level display of pointer variables.* Instead of displaying the pointer value to a section of memory, as most debuggers do, UDT displays the object pointed to by the pointer. For example, a pointer into a double linked list displays the entire record referenced by that pointer. The list can then be traversed forward by displaying the *next pointer* or backward by displaying the *previous pointer.*

*Debug procedures.* The user can create debug procedures in a language syntax identical to the language in which the application is written. These created procedures are compiled rather than interpreted to allow faster execution of the called procedure. The debug procedures can be executed on an event created by the program under execution (stopping at a breakpoint), or on a request by the user.

## 2.6. Human Interface

Testing and debugging tools are highly interactive software programs that deal with voluminous and complex information. An advanced human interface is critical in making such tools easy to learn and use. UDT employs display management through windows and item selection using a pointing device for its human interface. UDT provides the user with multiple views of the state of the subsystem under test through different windows.

Each window has a menu of commands. With these commands, the user can edit the information in that window via the keyboard or the pointing device. UDT windows can be manipulated by simple pointing device movements for effective screen management. The windows can be shrunk and expanded. The location of the windows can be physically moved to a different location on the screen. Open windows can be temporarily iconized to free the screen area for other windows. Display intensive windows can be scrolled both vertically and horizontally. The following sections describe the specifications of windows that UDT provides.

### 2.6.1. Command Window

The Command Window is the main window that is displayed upon invocation of UDT. Initially, it occupies the entire screen. It consists of a drop-down menu that lists the names of all other windows provided by UDT for displaying different views of the module under test. The last line of the Command Window is reserved for icons. This window displays the UDT syntax directed menu‡ and accepts the command line input from the keyboard. The syntax director does not submit a command for execution unless the command is syntactically valid.

### 2.6.2. Source Text Window

The Source Text Window displays high level source code of the module being tested. In addition, this window provides the user with capabilities to set and reset breakpoints and to select symbolic values directly from the source text for display or modification.

Hard and soft breakpoints can be set in the Source Text Window simply by moving the cursor to the desired statement number in the source code and clicking the pointing device. A single click

† Note that conditional breaks can cause the program to check a condition after each instruction is executed if a line number 'tag' is not present (e.g., cbrk when beta < 10 will cause check after each executed instruction).

‡ The UDT syntax directed menu displays a list of commands or command options that are relevant to the last token accepted by UDT. The syntax director enables the user to construct syntactically correct command lines.

signals a hard break, and a double click a soft break. In either case, the source code line at which the breakpoint was set is highlighted, indicating the type of breakpoint set. To reset a hard break or a soft break, the user moves the cursor to the desired breakpoint, and clicks the pointing device. The highlighting is removed to indicate that the breakpoint has been reset.

To display the values of variables during module execution, after any breakpoint event, the cursor is moved to an occurrence of the desired variable name in the Source Text Window, and the pointing device is clicked. A pop-up Dialogue Box appears on the screen and displays the name, address, and current value of that symbol. The user can change the current value of the symbol by editing the value field in the Dialogue Box. The Dialogue Box disappears when the user clicks on the Okay button with the pointing device.

The contents of the Source Text Window are dynamically updated during module execution. The update occurs after each step or breakpoint event. Text around a breakpoint is automatically displayed when the breakpoint event occurs. A marker indicates the location of the Program Counter. The position of the marker is also dynamically updated after each step or breakpoint event.

### 2.6.3. Breakpoints Window

The Breakpoints Window displays the current settings of all defined breakpoint registers. The user can add or delete breakpoint definitions by editing the fields in this window. The entries in the breakpoint table are dynamically updated during module execution. The update occurs when the user either sets or resets a breakpoint in the Source Text Window or at the command level. The update will also occur when a soft break is removed automatically by UDT when that breakpoint is reached during module execution.

### 2.6.4. Memory Window

The Memory Window is used for monitoring and displaying physical and symbolic memory contents. This window is characterized by a dictionary containing a set of memory partitions and symbol names. The dictionary entries are ordered by the time of definition of the partitions and symbol names. The user can add to or remove from the dictionary physical memory partitions or symbols. All the memory references entered in the dictionary, whether symbolic or physical, are displayed in the value fields of the Memory Window at the time of their definition. The user can modify any memory contents by physically overwriting the corresponding value fields. The value fields in the Memory Window are dynamically updated during any instance of module execution. The update occurs after each step or breakpoint event. Any change made to a symbol value in the Source Text Window or at command level is reflected in this window, if that symbol name is listed in the dictionary.

### 2.6.5. Register Window

The Register Window displays the current values of the machine registers as well as the value of the Instruction Pointer. The values of the registers are dynamically updated during any instance of module execution. The update occurs after each step or breakpoint event. The user has the ability to edit any of the register values, when the Register Window is active. Changes made to the register values at command level are automatically reflected in the Register Window. The values can be displayed or edited in octal, hexadecimal or decimal base.

### 2.6.6. Call Stack Window

The Call Stack Window displays the current chain of procedure calls in the module under execution. A sequence of fully qualified references to procedures are displayed in this window. Included with

each procedural reference is the parameter data types, their symbolic names, and their initial values upon entering the procedure. The reference listed first is the return address to which execution control will return when the current procedure exits. The second entry is the return address for the procedure that called the current procedure, and so on. The user is allowed to change the number of stack entries displayed in the Call Stack Window. The number specified can be such that either the top *n* entries (the *n* most recent procedure calls) or the bottom *n* (the *n* least recent procedure calls) of the stack are displayed. The entries in this window can be viewed symbolically or by values.

### 2.6.7. Application Window

The Application Window is used by the module under test to perform its input and output. The module's input and output is trapped by UDT and redirected to this window.

### 2.6.8. UST Window

The UST Window functions as a full screen editor, allowing the user to create and modify UST files.

## 3. Developing with UDT

Three main features compose the UDT testing facility: stubbing, assertion checking, and unit testing. The development example in this section illustrates the value of these features.

Assume an engineer is assigned the task of developing message passing utilities for a concurrent process application. The *job_controller* procedure creates a process as well as a unique mailbox associated with the process. The newly created mailbox is then placed into the post office (a data area that contains all mailboxes for all processes). Once the process becomes active, it can either send a message to another process, or read from its mailbox to see if another process had sent a message to it. Upon termination of the process, the mailbox must be removed from the post office.

The routines required for the message passing system include:

(i) a subprogram to create and add a mailbox to the post office
(ii) a subprogram to send messages to other processes
(iii) a subprogram to check a specific process' mailbox for messages
(iv) a subprogram to return an acknowledgement to the sending process if an acknowledgement is required
(v) a subprogram to delete a mailbox from the post office
(vi) other miscellaneous support routines for a message passing system

The procedure flow graph for creating a job is shown in Figure 5. Subprograms indicated in boxes are the routines that are part of the message passing utilities. Circled subprograms are the process control routines that have already been created in an earlier phase of development.

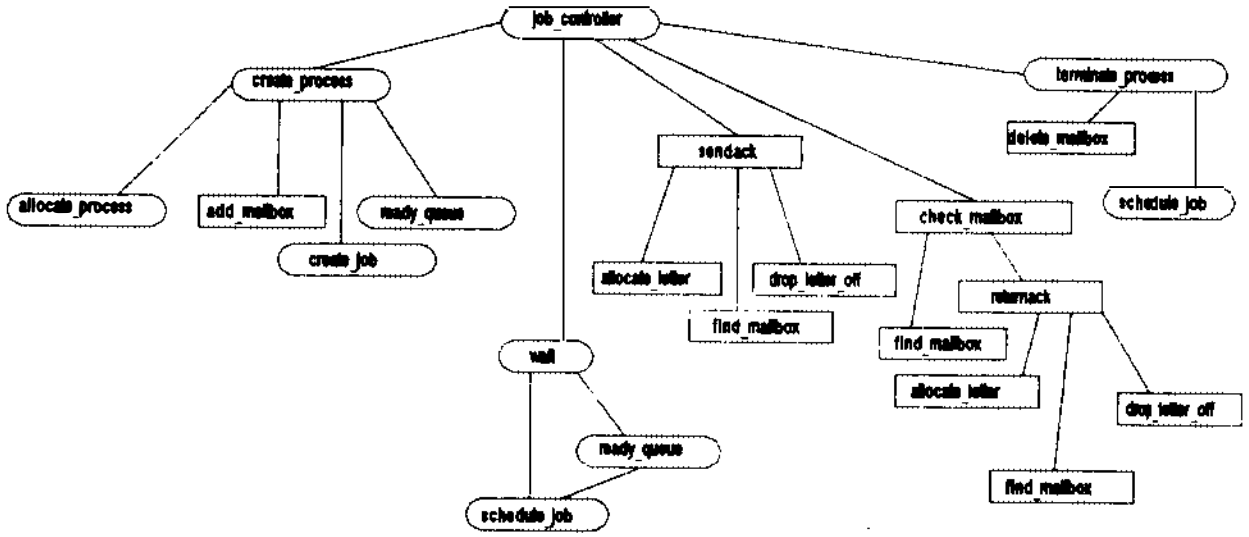



**Figure 5. CALL GRAPH**

### 3.1. Stubbing

To create a stub, the user first determines the possible entry conditions and corresponding exit conditions by examining the written requirements specifications. Next, the user creates a UST for the unresolved subprogram that is to be stubbed. Then, the user enables the stub for the unresolved subprogram by toggling the STUBBING field.

In this example, assume the subprogram *check_mailbox* is to be stubbed. The *check_mailbox* subprogram searches for the process' mailbox in the post office and reads any appropriate mailed messages. If a message requires a return acknowledgement, a letter must be allocated and returned to the sending process' mailbox. The stub for *check_mailbox* must simulate all these activities. Figure 6 shows the flow of procedure calls from *check_mailbox*. The stubbing feature can thus be used to mimic complex subprograms.

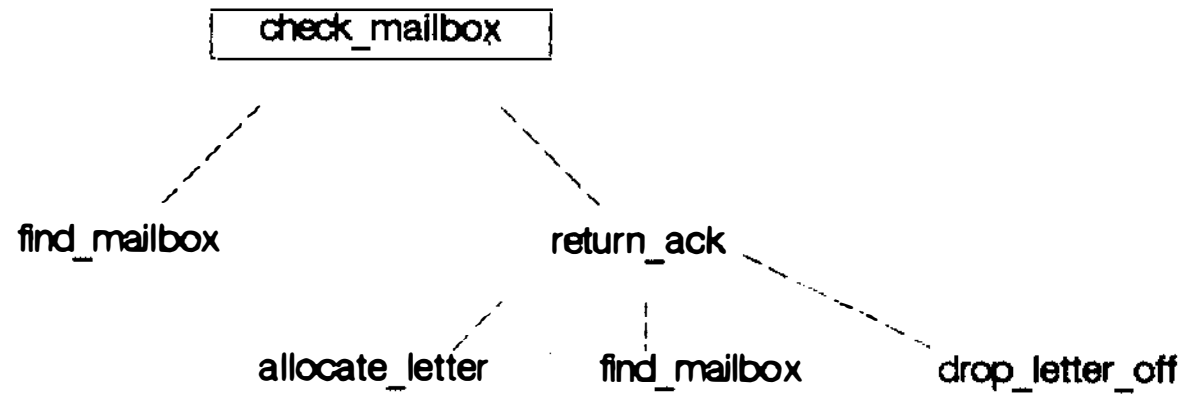



**Figure 6. STUBBING**

Based on the specifications, the user determines that the stub for *check_mailbox* can have several return conditions:

(i) The mailbox associated with the process cannot be found.
(ii) No messages are contained in the mailbox.
(iii) The message is processed.
(iv) The message is processed, and a return acknowledge is requested by the message.

From these determinations, the UST is created for *check_mailbox* shown in Figure 7.

SUBPROGRAM: check_mail_box ASSERTION: DISABLED STUBBING: ENABLED

| # | ENTRY CONDITION | EXIT CONDITION | STUB | ON ERROR |
|---|---|---|---|---|
| 1 | box_not_found(process_id) | | printf("mail box not found");<br>completion_code = -1; | |
| 2 | box.letters = = 0 | | printf("no messages);<br>return_address = "empty";<br>completion_code = 0; | |
| 3 | box.letters > 0 &&<br>box.mesg.recept< > retack | | return_address = "";<br>completion_code = 0; | |
| 4 | box.letters > 0 &&<br>box.mesg.recept= = retack | | message = "AK";<br>return_address = "process1;<br>completion_code = 0 | |
| 5 | | | | |

INVARIANTS: first_mailbox, current_mailbox

INPUTS: (char *) message, (char*) return_address

OUTPUTS: (char *) message, (char *)return_address, (int)completion_code

ON ERROR UPON ENTRY:

ON INVARIANT ERROR:

```
DEBUG PROCEDURES:
    box_not_found(process_id)
    char *process_id;
    {
       scanner = first_mailbox;
       while(scanner < > NIL) {
          if(strcmp(scanner - >id,process_id) = = 0)
             return TRUE
          else
             scanner = scanner - > next;
       }
       return FALSE;
    }
```

**Figure 7. UST SPECIFICATION FOR check_mailbox**

The user then runs the created specification table through the UST compiler to produce executable code used by UDT. The loaded subsystem is now ready to be instrumented. When the instrumented subsystem calls *check_mailbox*, control is passed to the compiled debug procedure provided by the created UST associated with *check_mailbox*. The entry conditions are checked and the appropriate exit condition is returned to the subprogram. Execution of the subprogram then

continues.

If a call is made to an unresolved subprogram that has no corresponding UST, then the program stops and UDT waits for the user to proceed. The user can create a UST for the unresolved subprogram and then proceed, or the unresolved subprogram can be ignored. If the user provides a UST with STUBBING enabled before giving the command to proceed, the debug procedure created from the UST will be executed. Otherwise, the call to the unresolved subprogram is ignored.

### 3.2. Assertion Checking

As in stubbing, in assertion checking the user determines the possible entry conditions and corresponding exit conditions for the subprogram to be asserted by examining the written requirements specifications. Then the user creates a UST for the subprogram whose interface is to be checked and enables the ASSERTION field.

In the example of the message passing system, assume that assertion checking is to be performed on *drop_letter_off*. Since *check_mailbox* is stubbed at this time, *drop_letter_off* is called only when a process sends a message to another process via *sendack*. Figure 8 shows the procedure flow graph for *sendack* (the interface to be checked is between *sendack* and *drop_letter_off*).

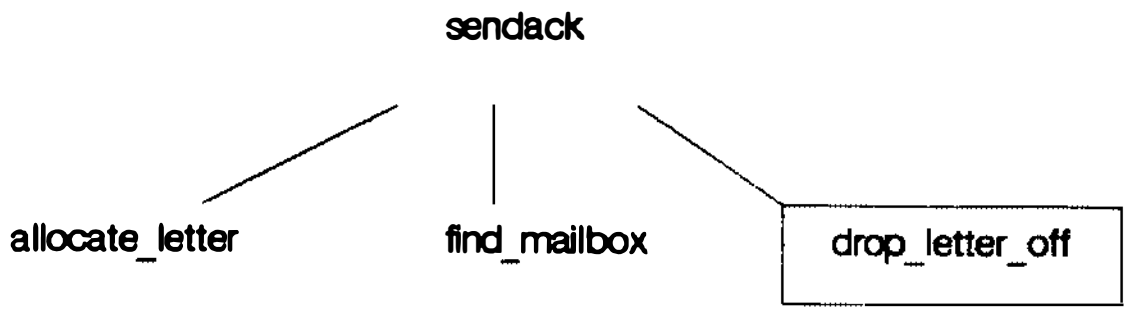



**Figure 8. ASSERTION CHECKING**

The engineer examines the written specifications for *drop_letter_off* and discovers that four parameters are passed to the routine:

(1) **mboxptr**: a pointer to a process' mailbox that will receive a message
(2) **newmail**: a pointer to the message itself.
(3) **priority**: the priority of the letter. If the priority is high, then the letter sent preempts all previous letters in the mailbox for the receiving process.
(4) **ackret**: a boolean to determine if an acknowledgement is requested by the sender of the message.

Further study of the specifications reveals that a mailbox can hold at most ten messages If the mailbox is full, then it is possible that messages can be lost. If it is empty (noted by a NIL pointer), then the message delivered becomes the first entry in the mailbox. Based on these given specifications, the engineer determines a series of entry conditions that may be valid when *drop_letter_off* is called. Also, from the specifications, the engineer knows the expected exit conditions of *drop_letter_off* for each corresponding entry condition. The engineer creates a UST for *drop_letter_off*. The partially completed UST for that routine is shown in Figure 9. After running the UST through the UST compiler, the engineer simply runs the subunit of code where *drop_letter_off* is called.

| SUBPROGRAM: drop_letter_off | | ASSERTION: ENABLED | | STUBBING: DISABLED |
|---|---|---|---|---|
| # | ENTRY CONDITION | EXIT CONDITION | STUB | ON ERROR |
| 1 | mboxptr = = "process4"&&<br>newmail < > NIL&&<br>priority = = STANDARD&&<br>ackret = = FALSE&&<br>letters_in_box_now< 10&&<br>first_slot< > NIL | last_slot = = newmail &&<br>letters_in_box_now < = 10 | | |
| 2 | mboxptr = = "process99"&&<br>priority = = URGENT&&<br>letters_in_box_now = = 10 | first_slot = = newmail&&<br>bumped_tail_msg = = TRUE | | show_box(mbxptr); |
| 3 | mboxptr = = "process4"&&<br>priority = = URGENT&&<br>letters_in_box_now < 10&&<br>first_slot = = NIL | first_slot = = newmail&&<br>letters_in_box_now< = 10 | | |
| INVARIANTS: | | | | |
| INPUTS: (char*)mboxptr, (struct letter *)newmail, (int)priority, (int)ackret, (int)letters_in_box | | | | |
| OUTPUTS: | | | | |
| ON ERROR UPON ENTRY: | | | | |
| ON INVARIANT ERROR: | | | | |
| DEBUG PROCEDURES:<br>show_box(mbox)<br>char *mbox;<br>{<br>printf("%s\n", mbox);<br>opsrc;<br>} | | | | |

**Figure 9. UST SPECIFICATION FOR drop_letter_off**

Suppose *sendack* makes a call to *drop_letter_off* with the following values as the actual parameters:

**mboxptr** = "process_4"
**newmail** = a legitimate message
**priority** = URGENT (highest priority)
**ackret** = TRUE

Also, the state of the application is such that the mailbox for "process_4" is empty:

**letters_in_box_now** = 0
**first_slot** = NIL

The UST will match these input values with the entry conditions of unit specification number 3 from the table in Figure 9. After the completion of *drop_letter_off*, the results are checked against

the exit conditions of unit specification number 3. The resulting outcome from the check is handled accordingly.

An important feature of UDT is that the user can run a subsystem with multiple stubs and assertion checks enabled. This feature greatly enhances the capability of the engineer to provide thorough exercising of completed subprograms at an early stage of development.

### 3.3. Unit Testing

As with the other forms of the testing facility, unit testing begins with a UST determined from written specifications. Entry conditions and expected exit conditions are written as unit specifications into a UST specific to the subprogram to be tested. After the UST is created, it is processed by the *Test Generator*, and a test suite is formed. The subsystem is then loaded and executed by the *Test Executor*. Depending on the mode of testing to be performed (*Raw Testing* or *Regression Testing*), UDT responds in a manner described in the Test Execution section of this paper.

As an illustration of both modes of testing, assume once again that the message passing scenario is used. The engineer must create a utility to add a mailbox to the post office when a new process is created. Knowing that the *job_controller* subsystem was created at an earlier stage of development, the engineer needs to add the routine *add_mailbox*, which is to be called from *create_process*. The engineer must also be careful not to introduce any faults or additional side effects to *create_process*. The calling procedure flow for this enhancement is shown in Figure 10.

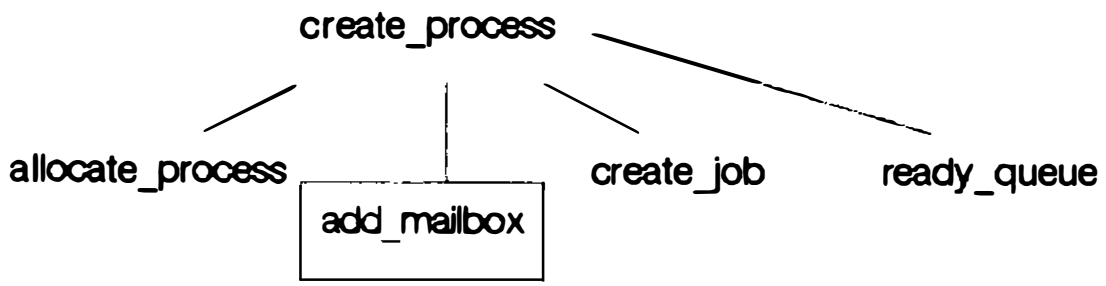



**Figure 10. UNIT TESTING**

After a slight code modification to *create_process* and the incorporation of *add_mailbox*, the engineer rebuilds the *job_controller* subsystem. Regression testing is performed by invoking the Test Executor and comparing the results in the Test Result Buffer from this run with the older (validated) Test Result Buffer. If no problems result from the comparison, the engineer can continue testing in raw testing mode. The UST is edited to allow more test conditions to exist in the table. The engineer adds the tests that will exercise the new code sequences created in *create_process* and *add_mailbox*. The test suite is regenerated by the Test Generator and the engineer then proceeds to test the newly created section of code. Once all the tests pass, a new Test Result Buffer becomes validated for future regression testing.

## 4. Future enhancements

Initially, UDT is targeted to run under UNIX* System V and to support object files with the Common Object File Format (COFF).

The eventual goal of UDT is to create a development environment whereby the engineer can develop an entire project with the support of the UDT system. If UDT is to become a complete development environment, an effort must be made to integrate the front end of development (requirements specifications, macro design, detailed design) with the back end (acceptance testing, bug tracking, product maintenance). To accomplish this goal, UDT must incorporate additional capabilities. These additional capabilities would be included in subsequent releases of UDT as enhancement packages. They include, but are not limited to, the following features:

*Increased spectrum of loaders and operating systems.* UDT should be able to debug and test applications written in most popular languages and under most popular systems. DOS is a strong consideration, as well as VMS.

*The ability to automatically create USTs from a formal specification tool.* As stated earlier, several attempts have been made to introduce formal languages for product specification. From one such specification language, it may be possible to parse the formal specification and produce USTs usable to UDT. A possible candidate would be a structured analysis-based design tool that could use USTs as mini specifications.

*Static analysis tools incorporated into the UDT system.* Static analysis tools that obtain metric values of the application can prove to be invaluable to the programmer. Program complexities, static syntax checks, static memory allocations, and boundary checks are some metric values that can provide useful data to a programmer.

*A line count and timing profiler.* This feature would provide a dynamic analysis to aid the programmer in determining algorithm efficiency as well as test coverage analysis. With the help of the profiler, the engineer will be able to focus attention on the performance bottlenecks or on sequences of code that have yet to be tested.

*Path coverage analysis.* The engineer will be able to identify all possible paths through the program and discover incomplete paths, or find dead code that is not reachable by any path. Using a path coverage analysis feature provides the engineer with further insight into the behavior of the program's execution, allowing the designer to modify inefficient algorithms and remove dead code.

*Function call graphs for specified sections of code.* Often, software projects are enormous undertakings, and it is not uncommon for programs to consist of many modules. Each of these modules, in turn, may be composed of numerous routines. A software engineer may spend many hours laboriously tracing the flow of procedure calls for complex programs. This feature automates the tracing of procedure calls and produces a road map of dynamic control flow for the engineer.

*Automated logging sessions for execution of USTs and automated bug reporting for failed assertions.* Logged test sessions provide a valuable source of information for a programmer who needs to know past testing practices. The logs provide all the sequences performed by the tester when the program was previously run. Automatic logging of assertion failures produces a record of failed test cases without causing the evaluator to interrupt the test session to log the error manually.

*Connection to a centralized data base.* A centralized data base to contain the entire collection of USTs and a configuration management system to control the access of information would be a great

*UNIX is a registered trademark of AT&T.

asset to UDT. The development team would have easy access to a localized library of test suites, test result buffers, and a test generation function library for a particular project.

## 5. Summary

UDT is a state-of-the-art software tool that provides a very practical solution to the problem of early system testing. However, much experimenting and user feedback is still required to prove its claims. Some of the concepts introduced in this paper may be reviewed, refined or modified during project development. Our expectations are that UDT will greatly contribute to:

(i) Reducing software development time, by *catching bugs early*
(ii) Improving the quality of delivered software products by *integrating the testing and debugging phases with the detailed product specification phase.*

A software engineer can further benefit for the following reasons:

(i) UDT provides a formal framework to ease communication between software system designers, implementors and evaluators.
(ii) UDT's formal unit specification language allows a developer to succinctly define and communicate the specifications of software units to other developers/testers.
(iii) UDT eases product maintenance by allowing modified parts to be regression-tested *individually.*

UDT is not just another debugger. We believe that its integration with front end tools (such as design tools) and with back end tools (such as in circuit emulators and performance analyzers) will make UDT a very important functional bridge between the conception of a software product and its delivery.

## 6. Acknowledgements

Many thanks are owed to Garry Thompson of Intel Development Systems. His invaluable help, dedicated effort, and creative ideas helped sculpture some of the concepts found in UDT. We are grateful for his work on the project.

## 7. References

(1) Dijkstra, Edsger W., "Guarded Commands, Nondeterminancy and Formal Derivation of Programs," *Communications of the ACM*, Vol. 18, No. 8. August 1975, pp. 453-457.

(2) Gehani, N. and A. D. McGettrick editors, *Software Specification Techniques*, Addison-Wesley Publishing Company, Menlo Park, 1986, 477 pp.

(3) Gries, David, *The Science of Programming*, Springer-Verlag, New York, 1981, ch. 6, pp. 99-105.

(4) Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, Vol. 12, No. 10. October 1969, pp. 576-583.

(5) Hood, Robert T. and Ken Kennedy, *A Programming Environment for Fortran*, Published by the Department of Computer Science, Rice University, Rice COMP TR84-1, June 1984.

(6) Hurley, Richard B., *Decision Tables in Software Engineering*, Van Nostrand Reinhold Company Inc., New York, 1983.

(7) Johnson, John D. and Gary W. Kenney, "Implementation Issues for a Source Level Symbolic Debugger (Extended Abstract)," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Order No. 593830, 1983, pp. 149-151.

(8) Müllerburg, Monika A. F., "The Role of Debugging Within Software Engineering Environments", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Order No. 593830, 1983, pp. 81-87.

(9) Pressman, Roger S., *Software Engineering: A Practitioner's Approach*, MaGraw-Hill Book Company, San Francisco, 1982, ch. 10, pp. 249-252.

(10) Seidner, Rich and Nick Tindall, "Interactive Debug Requirements," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Order No. 593830, 1983, pp. 9-22.

# UDT
# Unit Debugger and Tester

## AGENDA

(1) What is UDT?

(2) The UDT Approach

(3) Examples of Use

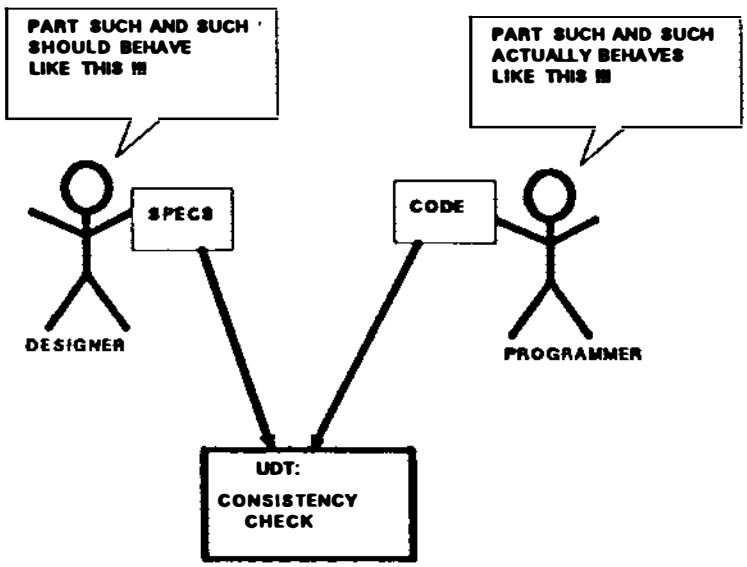
(4) How Does UDT Help Project Development?

(5) Summary

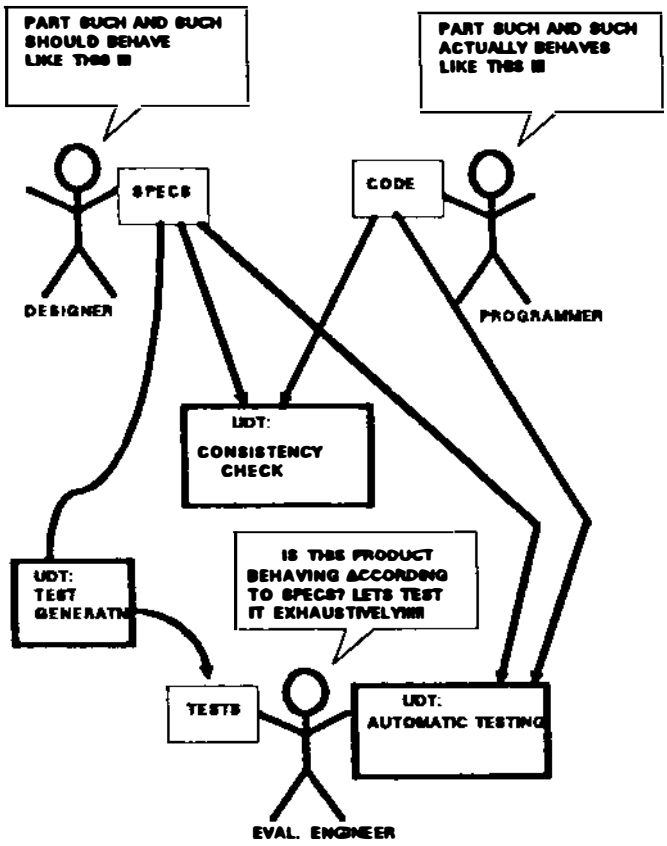
## WHAT IS UDT ?

- A debugger with a window oriented human interface.
- A system that allows parts of a software product to be debugged or tested in isolation.
- A tool that provides a more formalized method of defining a software product through a specification language.
- A system that checks specifications against the actual behavior of the software product at run time.
- A system that generates test suites from the software product's specifications.
- A system that provides a stubbing facility to evaluate software product prototypes.

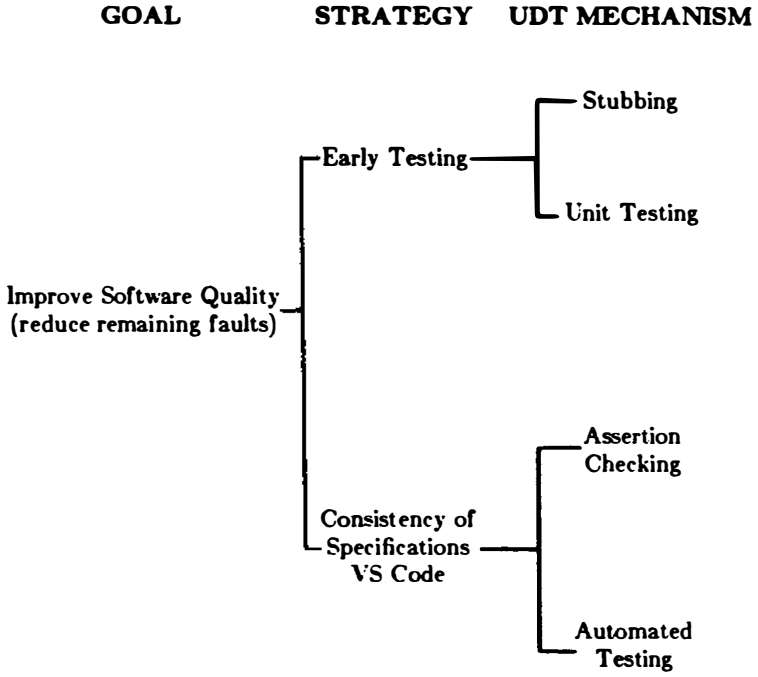
## UDT: EARLY BUG DETECTION

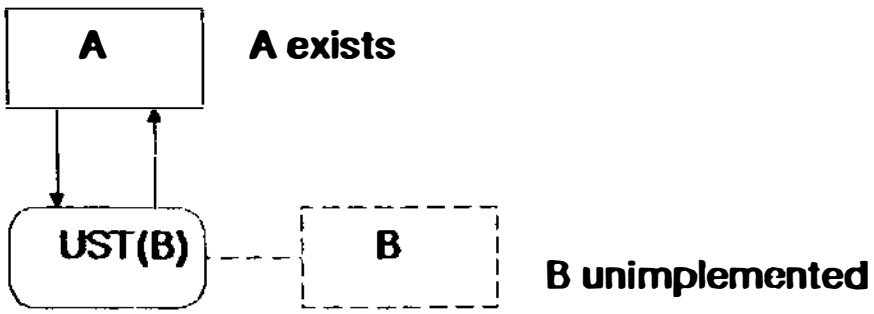


# UDT: AUTOMATED TESTING

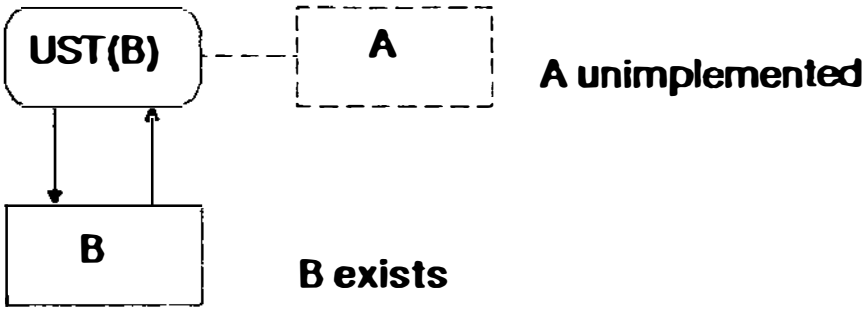



## THE UDT APPROACH

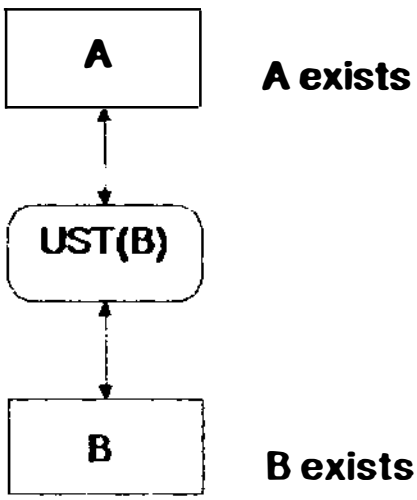





STUBBING

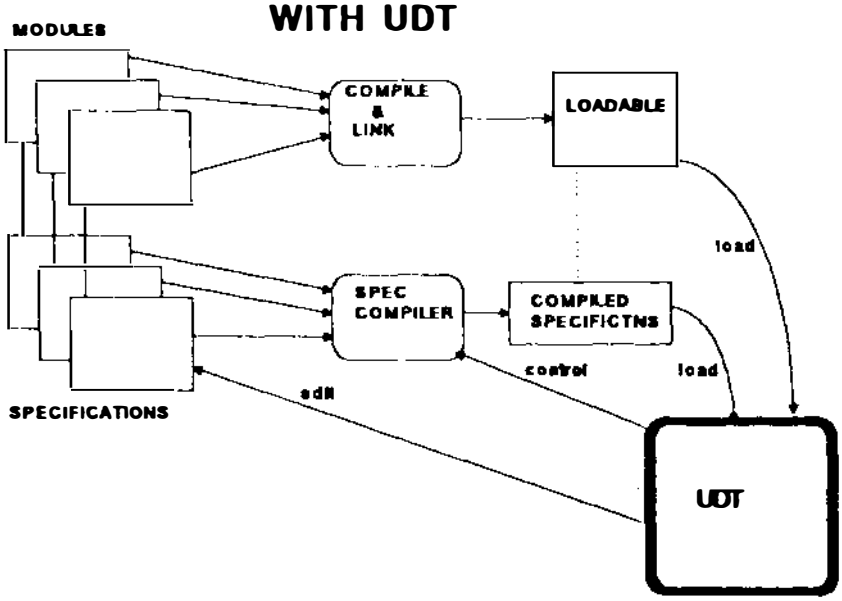



UNIT TESTING




ASSERTION CHECKING

## DEVELOPING SOFTWARE WITH UDT




## EXAMPLES OF USE

Two examples will be shown.

(1) The STUBBING Feature

(2) The ASSERTION CHECKING Feature

# SPECIFICATION

put_node( subtree, new_node )

```
    nodeptr  subtree;
    nodeptr  new_node;
```

returns (int) ret_val.

Given the following inputs:

- a pointer to a position in a binary tree (**subtree**)
- a node to be added to the tree (**new_node**)

Place the node in a sorted fashion (alphabetically) into the binary tree. If there is no tree in existence, the node shall become the root. If the node already exists, then produce a message indicating so and return without placing the node into the tree.

**ret_val** is assigned the value 1 if the placement of the node is successful.
**ret_val** is assigned the value of -1 if the node already exists in the tree.

| SUBPROGRAM: put_node | ASSERTION: ENABLED | | STUBBING: DISABLED | |
|---|---|---|---|---|
| # | ENTRY CONDITION | EXIT CONDITION | STUB | ON ERROR |
| 1 | subtree == NIL | root == new_node && ret_val == 1 | | |
| 2 | subtree <> NIL && exists(new_node) | ret_val == -1 | | |
| 3 | subtree <> NIL && strcmp(new_node→name, subtree→name) < 0 | subtree→left→name == new_node && ret_val == 1 | | |
| 4 | subtree <> NIL && strcmp(new_node→name, subtree→name) > 0 | subtree→right→name == new_node && ret_val == 1 | | |

**INVARIANTS:**

**INPUTS:** (nodeptr) subtree; (nodeptr) newnode;

**OUTPUTS:** (int) ret_val;

**ON ERROR UPON ENTRY:** entry_error(ABORT)

**ON INVARIANT ERROR:**

**DEBUG PROCEDURES SECTION:**

```
struct node {
        char    name[70];
        char    address[70];
        char    telno[10];
        int     deleted;
        struct node  *left_child;
        struct node  *right_child;
}

exists( new_node )
  struct node  *new_node;

{
  struct node  *enode;

  if ( (enode = tree_search(root, new_node→name)) == NIL )
    return (FALSE);
  else
    return (TRUE);
}

tree_search( subtree, name )
  struct node  *subtree;
  char        *name;
```

# SPECIFICATION

```
get_a_token( line, max, f )
     char  line[ ];
     int   max;
     FILE  *f;
```

returns (TOKENTYPE) toktype.

Given a pointer to a file (**f**), an empty token buffer (**line**), and the **max** size of an allowed token, read a line in the file pointed to by **f** and fill the empty token buffer **line**. The number of characters allowed in the token buffer cannot exceed **max** size. Return the token buffer (now full) and its token type **toktype** to the caller.

| SUBPROGRAM: get_a_token | | ASSERTION: DISABLED | | STUBBING: ENABLED |
|---|---|---|---|---|
| # | ENTRY CONDITION | EXIT CONDITION | STUB | ON ERROR |
| 1 | TRUE | | make_token() | |

INVARIANTS:

INPUTS: (Int) max; (FILE *) f; (char *) line;

OUTPUTS: (char *) line; (TOKENTYPE) toktype;

ON ERROR UPON ENTRY:

ON INVARIANT ERROR:

DEBUG PROCEDURES SECTION:

```
int   req_cnt = 0;  /* global declaration for this UST */

make_token()

{
  char  token[30];
  static struct {
              int  req_tok;
              char *req_str;
  } request[] = {
       CREATE_RECORD,    "CREATE",
       NAME,             "P. Opus",
       ADDRESS,          "Bloom County",
       TEL_NO,           "(105) 222-3344",
       CREATE_RECORD,    "CREATE",
       NAME,             "Steve Dallas",
       ADDRESS,          "Bloom County",
       TEL_NO,           "(105) 222-7765",
       CREATE_RECORD,    "CREATE",
       NAME,             "Bill the Cat",
       ADDRESS,          "Russia",
       TEL_NO,           "Classified",
       END,              "End of List"
    };

  strcpy( token, request[req_cnt].req_str );
  line = token;
  toktype = request[req_cnt].req_tok;
  req_cnt++;
  return (TRUE);
}
```

## HOW DOES UDT HELP PROJECT DEVELOPMENT ?

- UDT reduces software development time ***by catching bugs early***.
- UDT improves the quality of the delivered software product by *integrating the testing and debugging phases with the detailed product specification phase*.
- UDT provides a formal framework to ease communication between software system designers, implementors, and evaluators.
- UDT's specification language allows an engineer to succinctly define and communicate the specifications of software units to other developers/testers.
- UDT eases product maintenance by allowing modified parts to be regression-tested *individually*.

## SUMMARY

- UDT provides a practical solution to the problem of early testing.
- UDT can improve engineering productivity.
- UDT provides a formalized method of specifying software units, thus resulting in a more clearly defined product.
- UDT supports development, testing, and maintenance of a software product.

# TCL and TCI, A powerful language and an interpreter for writing and executing black box tests

Arun Jagota
Vijay Rao

Software Engineering Group
Intel Corporation

This paper describes a language called TCL (Test Case Language) and an interpreter for it called TCI (Test Case Interpreter). TCL is a very simple but powerful language which allows rapid development of test cases. Test cases can be entered interactively or read in from a file. TCI has been implemented and is currently being used to test Intel's iRMX family of Operating Systems.

The first section discusses the motivation for this project. The next two sections describe the salient features of the language TCL and its interpreter TCI. These sections are then followed by a brief description of an implementation of TCI. The following section describes some experiences that the authors have had in using this tool. The final section discusses briefly the future directions for this tool.

## Biographies

Arun Jagota is currently working in the Software Evaluation Group at Intel Corporation, Hillsboro, Oregon. He has a Bachelor Of Technology in Electrical Engineering from Indian Institute Of Technology, Delhi, India and a Master Of Science in Computer Science from University Of Kansas, Lawrence. His current interests include operating Systems, language processors and artificial intelligence.

Vijay Rao is currently working in the Multibus II Engineering Group at Intel Corporation, Hillsboro, Oregon. He has a Bachelor Of Technology in Electrical Engineering from Karnataka Regional Engineering College, India and a Master Of Science in Computer Science from University Of California, San Diego. His current interests include operating systems, multiprocessor systems and software engineering tools.

---

## 1. Motivation

The cost of test case development, test case maintenance and interpretation of test results is a very significant part of the cost of a large software project. Moreover, if the software is developed in phases, then a regression test base has to be established and executed before making the transition from one phase to the next [4].

TCL was developed to allow efficient construction and maintenance of test cases and automatic verification of results. In the short term, TCL is beneficial in reducing test case development time by making it very easy to create test cases to verify the functional and performance specifications of a large software program.

Consider writing a simple test for a procedure P exported by a very large multi module program S. Normally, the test would first be coded in a high level language, compiled, linked in with S - the program to be tested and then executed. To make slight changes to the test, one would have to repeat all the steps again. This imposes a high test development overhead in an environment where tests have to be rapidly constructed. In TCL, the above test case would be coded as a single line

P (<Param_List>)

where P is the name of the procedure to be tested and <Param_List> is the list of actual parameter values to be used. This implies that small test cases can be created and interactively executed almost

instantaneously, thus also making it very simple to perform "Random" or "What - IF" kinds of tests. The above scheme is possible only because TCI can be made to run in the environment of the program to be tested. The details of how this is implemented are described in a later section.

## 2. The Language

The primary focus of TCL is to provide a compact form in which a procedure to be tested can be invoked with different values for each parameter and the expected output specified for each execution instance. Values returned by the procedure under test can be assigned to identifiers so that they are available for future use.

We are going to describe the language TCL with the help of examples. Let us start with some simple examples first and then graduate to more complex examples,in the process, covering most of the important features of TCL.

In order to do this, let us first define a function called Test_Proc which will serve as the procedure under test for all our examples. Test_Proc takes in three input parameters and returns their product. Test_proc also returns an exception which has one of the following values.

(1) NORMAL - If all the input parameters are greater than 0.
(2) NEGATIVE - If at least one of the input parameters is less than 0.
(3) ZERO - If at least one of the input parameters is equal to 0 and none of the input parameters is less than 0.

A simple invocation of Test_Proc could be

```
Test_Proc (5,6,8)
```

where 5, 6 and 8 are the values for the first, second and third parameters respectively. TCI will execute this call and report its output in a format that will be described later.

Suppose we want to repeat the above procedure call 5 times. This can be done as follows.

```
Test_Proc (5,6,8) 5
```

Next suppose that we want to execute Test_Proc with three different values for the first parameter. This can be done as follows.

```
Test_Proc ({1,2,3}, 6, 8)
```

Test_Proc will be executed thrice, with the first parameter values as 1,2 and 3 respectively.

Now we can also store the result of the function call into an identifier as shown below.

```
Test_Proc , I1 (5, 6, 8)
```

In this example, the result is stored back into the identifier I1.

In fact, we can perform multiple invocations of a procedure (with different parameter values) and store the results back from each execution through a single command as shown below.

```
Test_Proc , I1-I3 ({1,2,3}, 6, 8)
```

In the above example, the results of the three executions will be stored in I1, I2 and I3 respectively. If we wanted, we could then use I1 to I3 as input parameters to Test_Proc as shown below.

```
Test_Proc (I1-I3,6,8)
```

Test_Proc will be executed thrice, with the first parameter values being I1, I2 and I3 respectively.

Actually, another way of storing results into identifiers and reusing them later is by means of what are known as aggregates. Aggregates store the results of all the invocations from one command into a composite identifier. Any references to that identifier in future implies that all the values associated with that identifier will be used. The above two examples can be re-written to use aggregates as shown below.

```
Test_Proc, I* ({1,2,3}, 6, 8)
Test_Proc     (I*, 6, 8)
```

One very important feature of TCL is its support for boundary value testing. In order to demonstrate this feature, we need to modify the specification of the function Test_Proc to the following. Let us say that the legal values for the first parameter to Test_Proc are in the range 1 to 1000. If the first parameter value is in this range, then the exception is "NORMAL". If the first parameter value is < 1, then the exception is either "ZERO" or "NEGATIVE". If the first parameter value is > 1000, then a new exception called "LIMIT" will be generated. The following example shows how we can write in TCL, a boundary value test for the first parameter of Test_Proc.

```
Test_Proc (1..1000, 6, 8)
```

The ".." implies that the corresponding expression is a boundary value range. A procedure using a boundary value range is invoked with the following values associated with that range.

- The two bounds of that range.
- The two neighboring values around each bound (upper and lower).
- N values uniformly selected from between the bounds. N can be specified as an attribute of the range or a default of three will be used.

In the above example, Test_Proc will be invoked 9 times with the following values for the first parameter.

```
0, 1, 2, 250, 500, 750, 999, 1000, 1001
```

Another very important feature of TCL is that execution instances can be generated based on the cross product of the list of parameter values. The following example illustrates this feature.

```
Test_Proc (1-3, {2, 4, 6}, 8):C
```

This command will invoke Test_proc 9 times with the following sets of parameter values.

```
P1   1 1 1 2 2 2 3 3 3
P2   2 4 6 2 4 6 2 4 6
P3   8 8 8 8 8 8 8 8 8
```

Of course, if we want to store the results of each of these executions into identifiers, then this can also be accomplished as follows.

```
Test_Proc ,I1-I9 (1-3, {2, 4, 6}, 8):C
```

Consider a similar command, minus the ":C" at the command tail.

```
Test_Proc (1-3, {2. 4. 6}, 8)
```

Instead of generating the cross product, for the nth execution. TCL uses the nth value from the list of values for each parameter. If a list associated with a particular parameter has less than n values. then the last value in the list is used. In this particular example, the sets of parameter values for the different executions will be

```
P1   1   2   3
P2   2   4   6
P3   8   8   8
```

It is not necessary to assign values to all parameters in a command invocation line. Default values will be used for all parameters left unspecified. The following examples describe how parameters may be left unspecified.

```
(1)   Test_Proc
(2)   Test_Proc (5,,8)
(3)   Test_Proc (5)
```

In the first example, default values will be used for all the parameters. In the second example, the default value will be used for the second parameter. In the third example, default values will be used for the second and third parameters.

So far, in our discussion, we have only seen parameters which use simple numeric values. In addition to such parameters, TCL also supports strings and structured parameters. In order to demonstrate their use,

let us alter the specification of Test_Proc one more time. Let us give Test_Proc four more parameters - P4, P5, P6 and P7 which are defined as follows.

(1) P4 - input parameter which accepts values of type STRING.
(2) P5 - input parameter which accepts values of type ARRAY OF NUMBERS.
(3) P6 - input parameter which accepts values of type ARRAY OF STRINGS.
(4) P7 - output parameter which returns a STRUCTURE which has the following fields.
    - F1 - Numeric value.
    - F2 - String value.

The following example describes an invocation of the new Test_Proc.

```
Test_Proc  (5, 6, 8, 'STRING', A(1, 3, 5), A('AA', 'SS'),S[I1, S1])
```

In this example, the fifth parameter is an array with three elements whose values are 1, 3 and 5 respectively. The sixth parameter, similarly, is an array of two string elements. The seventh parameter is an output STRUCTURED parameter. The value of the first field is to be stored into I1. The value of the second field is to be stored into S1 which is an identifier used for storing strings.

### 2.1. Specifying expected output

Expected output can be specified for each test written in TCL, allowing results to be verified automatically. TCL divides the specification of expected output into three components.

- Expected errors(exceptions) - (For negative testing).
- Expected values of output parameters.
- Expected performance (actual performance should be <= expected performance).

Let us take the first example that we used with Test_Proc and associate expected output with it.

```
Test_Proc  (5,6.8)
$expected_output
exception       = NORMAL
returned_value = 210
$end_output
```

This implies that we expect Test_proc to return a "NORMAL" exception and return the number 210 as its function value.

Let us now describe a more complex example in which expected output can be specified for all the invocations of a multiple invocation command.

```
Test_Proc  (1..1000,6,8)
$expected_Output
exception       = {ZERO, NORMAL*7, LIMIT}
returned_value = {, 48, 96, 12000, 24000, 36000, 47952, 48000}
$end_output
```

This implies that the expected exception from the first invocation is "ZERO", from the next seven invocations is "NORMAL" and from the last invocation is "LIMIT". No particular return value is expected from the first execution, the expected return values from the next seven invocations are the ones listed in sequence and no particular return value is expected from the last execution.

If any of the expected values does not match the corresponding actual value then the test has "FAILED". If a test fails, then the execution instance(s) which failed are recorded so that they can be recreated later.

## 2.2. Remote Procedure Execution

TCL greatly simplifies creating functional test cases for multi-processing (multi-tasking) programs by providing a master environment from which tests can be initiated for remote execution from other processes (tasks). For example, we can say something like

<Execute procedure P1 from process (task) PR1>
<Execute procedure P2 from process (task) PR2>

The user is responsible for first creating the process (task) from which a procedure is to be executed remotely. He is then allowed to mention this process (task) in the command line for the procedure invocation. If a particular process (task) is not explicitly specified, then the procedure is executed from the context of TCI.

In the following example, a task will first be created. That task will then be specified as the remote execution point in a subsequent command.

```
create_task, t1    – Create a task and return its identifier in t1
Test_Proc ,,t1     – Perform the procedure Test_Proc from the
                   – task t1
```

There are two ways in which procedures may be executed remotely - serially or concurrently. In the serial mode, all the remote procedure executions are performed one by one. In the concurrent mode, successive remote procedure execution requests are forked off to different processes and TCI waits for all of them to complete before processing any further commands. The following example illustrates the serial mode.

```
Create_task, T1
Create_task, T2
Create_task, T3
$SERIALISE
Test_Proc,, T1  (1,2,3)
Test_Proc,, T2  (4,5,6)
Test_Proc,, T3  (7,8,9)
```

In this example, the three tasks are first created and their ids are put in T1, T2 and T3 respectively. The first Test_Proc is then executed from the task T1. After it completes execution and its results are processed, the second Test_Proc is executed from the task T2 and finally, the third Test_Proc is executed from the task T3.

The next example illustrates the concurrent mode.

```
Create_task, T1
Create_task, T2
Create_task, T3
Test_Proc,, T1  (1,2,3)
Test_Proc,, T2  (4,5,6)
Test_Proc,, T3  (7,8,9)
```

In this example, the three tasks are first created and their ids are put in T1, T2 and T3 respectively. The three Test_Proc calls are then shipped simultaneously for execution from the three tasks T1, T2 and T3. TCI waits for each of them to complete before processing any more commands. The order in which these procedure requests are handled depends on the operating system on which TCI is hosted.

### 2.3. Directive Commands

TCL also has quite an extensive list of built in utility functions.

(1) **Timing Functions:** As mentioned earlier, TCL can be used to measure the performance of procedure invocations. When the "TIME" function is invoked, all subsequent commands are timed until a "NOTIME" function is invoked. A stop watch option is also provided which measures the cumulative time between when a "TIME" is invoked and when a "NOTIME" is invoked.

(2) **Memory Management functions:** TCL has commands for initialising segments of memory, comparing two segments of memory and displaying segments. These segments can be used to test procedures which manipulate memory buffers ( for example i/o procedures like read, write and message passing procedures like send, receive).

(3) **Output reporting functions:** TCL has commands for selecting the mode in which numeric values are output (hexadecimal or decimal). It can also be told whether to print the output for all tests or only those tests that failed.

(4) **Remote procedure execution mode:** TCL allows specification of whether remote procedures should be executed serially or concurrently.

(5) **Script management functions:** TCL has functions which make it easy to combine a lot of disjoint test cases into one large script. TCL allows each test case to be followed by a reset function so that all identifiers and the state of TCI are reset to the original values. TCL also allows the beginning and end of each test case to be marked with the key words BEGIN and END respectively. If an error occurs inside one test case, then TCI will ignore the rest of the commands in that test case and proceed to the beginning of the next test case.

(6) **Comments and white space:** TCL allows comments to be interspersed with commands. It also allows liberal use of tabs and white space.

(7) **Listing control functions:** TCL itself does not provide listing control functions. But it allows scripts to be formatted by a text formatter called Sanscribe by skipping all lines which contain Sanscribe commands. Sanscribe commands support items like page headers, footers, page breaks, section headers etc which can as a result be used in TCL scripts.

## 3. TCI and the run time environment

TCI is an interpreter for TCL. It executes TCL commands and also formats and reports the output. TCI can be used in interactive sessions, that is execute TCL commands entered from a terminal and report results back to the terminal. TCI can also interpret commands from a script file and report results back to another file.

### 3.1. Reporting results:

For each command which executes a procedure under test, the following items of information may be reported - the line number of the TCL command in the script, the procedure call(s) and all the actual values that were used for it's parameters, the number of times this procedure was invoked and the invocation instances which failed (if any). The actual output is then reported as follows.

(1) The exception name returned by each invocation of the procedure.
(2) The execution time of each invocation of the procedure.
(3) The values of the output parameters for each invocation of the procedure.

The following example describes the output for the following command. Test_call is a function returning the product of three numbers.

```
Test_Proc  ({1,3,5},{2,4,6},{3,5,7})
```

– Output

```
- Source Line # 1 -
Test_Proc  ({1,3,5},{2,4,6},{3,5,7})
Number of Executions = 3
```

```
Exception          = {NORMAL*3}
Return_value       = {6,60,210}
```

We may observe that since the returned exception is the same for all the three instances, it is concisely represented by "NORMAL*3".

A more complex example illustrates much better the concise form in which the output is reported.

```
Test_Proc   (0-1,0-1,1-20):C
```

– Output

```
- Source Line # 1 -
Test_Proc  ({0*40,1*40},{0*20,1*20,0*20,1*20},{1-20,1-20,1-20,1-20})
Number Of executions = 80
Exception          = {ZERO*60,NORMAL*20}
Returned_value     = {*60,1-20}
```

### 3.2. Run time environment for TCI:

TCI runs in the environment of the program to be tested. Before any test cases can be executed, the names of the procedures to be tested have to be cataloged in a table provided by TCI. TCI has then to be linked in with the program(s) to be tested.

**Tables:** TCI also maintains two other tables - an exception table and a constant table. The exception table stores the names of all the exceptions (errors) that can be returned by procedures under test. The constant table is used to associate names with constants. Both these tables are fairly easy to create and can be linked to TCI.

**Driver procedures:** For each procedure to be tested by TCI, a driver procedure has to be written. This driver procedure is responsible for acquiring the actual parameter values from a standard template, formatting the parameters for the actual procedure invocation, making the call and setting the return values back into the same parameter template. Also associated with each procedure to be tested is a data structure which stores information about the parameter types of that procedure. This is used by TCI to do semantic checking of the TCL command line.

All the driver procedures and procedure data structures are contained in separate module(s) and linked in with TCI.

**Remote procedure execution:** Remote procedure execution is based on message passing. Commands to be executed remotely are shipped as messages to the appropriate process. The corresponding process waits to receive the message, executes the command and sends the results back to TCI in another message. After having shipped a command, TCI waits to get the results back and then compares them against expected results or formats them for reporting or both. Since the message passing mechanism is dependent upon the Operating system on which TCI is hosted, the remote procedure execution part of TCI is contained in a separate module. This OS dependent module can be linked in with TCI making it easy to host TCI on different Operating systems.

**Memory management features:** As mentioned earlier, TCI also provides mechanisms for initialising, displaying and comparing segments of memory. These segments can be used to test procedures which manipulate memory segments (buffers). Since memory management is also dependent on the Operating system on which TCI is hosted, these features are contained in a separate module which can be linked in

with TCI.

## 4. Implementation

TCI has been implemented to run on Intel's iRMX family of operating systems. It is written in PL/M and uses an operating system independent interface [3]. This has facilitated it to be ported to Xenix with almost no changes to the source code. The definition of TCL and the implementation of TCI took about eight man months.

**Scanner and Parser :** The Scanner breaks the input stream into tokens which are passed to the Parser. The Scanner also ignores blank lines, spaces and comments. The Parser is of Recursive Descent type, and is responsible for syntax analysis and passing the procedure to be tested and its parameters in the form of a record to the Parameter Handler. It is implemented as a set of procedures for each Non-Terminal of the grammar [2]. The Parser maintains the identifier tables and passes the actual values of the identifiers to the Parameter handler. If defaults are used then they are looked up and passed to the Parameter Handler.

TCL is for the most part a regular expression language [1]. This simplifies the implementation of the parser to a fair extent. One interesting feature of TCL is that in many places we found grammers for two parts to be so similar that we could merge them into a single grammar by using parameters.

**Error handling :** The Error Handler prints in a standard format, errors encountered during scanning, parsing or parameter handling stages. The line number in the input file, the type of symbol encountered and the type of symbol expected is printed for syntax errors. TCI can recover from syntax or semantic errors in one command and continue processing further commands. If errors are caused because of internal problems (such as running out of memory) an error message is printed and TCI aborts.

**Parameter Handler :** The Parameter Handler accepts the procedure record from the Parser and produces another record which is passed to the procedure driver. It is responsible for handling boundary and enumerated values and repetition of the test procedure. The Parameter Handler also handles the returned values from the test procedures and does the assignment of these values to identifiers if necessary.

The heart of the parameter handler is an Abstract Data Type called "Set". This abstract data type has the following operations associated with it - read(set), write(set), create(set), close(set), reset(set). As an example, each list of values associated with a particular parameter in a procedure invocation command is stored in a variable of type set. The parameter handler can then read the next value from a "set" variable, write a value into a "set" variable or manipulate the "set" variable in other ways using the above mentioned primitives.

The following three are the fundamental tasks performed by the parameter handler which are based directly on the abstract data type set.

(1) Selecting the values to be used for each parameter for the next invocation instance.
(2) Selecting the expected output values to be used for comparing against the results of the current invocation instance.
(3) Inserting the results of the current invocation instance into the output buffer. The output buffer will be printed once the results of all the invocation instances have been tabulated.

**Driver :** The Driver consists of a set of procedures, each of which, actually makes the test procedure call. It receives from the Parameter Handler, a parameter block with a pointer to its parameters, a flag if the procedure call has to be timed and space for the return value, the exception value and the timing. If timing of the test procedure is needed, it will be done by the driver procedure.

The address of the driver procedure, its parameter count, and its mnemonic have to be inserted into a table along with its default parameter values if a new call is added. Adding new test procedures is easy since each driver procedure receives a standard parameter record.

## 5. Experience

TCI and TCL have currently been used in the functional and performance testing of system calls in the iRMX family of operating systems. A test base has been established for system calls in the Kernel and the I/O system [3].

The facility to perform timing measurements on test procedures has helped in merging the performance and functional tests. Previously there used to be separate functional and performance tests for the system calls. This has helped in reducing the maintenance costs of the tests.

The incremental work needed to test new system calls has also been reduced and this facilitates rapid development of test cases for new system calls.

## 6. Conclusion and Future Directions

TCL and TCI have shown that an automated testing tool can reduce both the test development and test execution time. Since tests can be rapidly developed, this implies that TCI is also well suited for use as a unit testing tool. One can also use TCI as a human interface to, for example, Operating systems, by building it to execute Operating systems calls.

## 7. References


(1) Aho and Ullman. *Principles Of Compiler Design.* Addison-Wesley, April 1979.

(2) Wirth, Nicklaus. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

(3) Intel Coproration. *iRMX 86 System Call Reference Manual.* Hillsboro, Oregon.

(4) Myers, Glenford. *The Art Of Software Testing.* John Wiley And Sons, 1979.

# TCL/TCI

- TEST CASE LANGUAGE/INTERPRETER





## Tests in a conventional programming language

- Find, define, and maintain external definition files for the procedures to be tested
- Code logic for performing procedures from different contexts.
- Initialize (Open) reporting (Output) mechanism.
- Code actual procedure (to be tested) and all of its parameters.
- Code logic for checking expected return and output values.
- Code logic for reporting deviations.
- Code logic for skipping ahead if previous tests fail.
- Create scripts for linking the test with the program to be tested

intel

## Tests In TCL

- Code procedure name
- Optionally overide default parameters
- Optionally specify expected output

TCL



intel

## TCL Features

- Free form script language.
  - Enumerated parameter values
  - Cross product notation
  - Black box parameter values
  - Integer,string,structure notation
  - Literals
  - Compact and concise syntax
- Context for Processes/tasks
- Optional specification of EXPECTED RESULTS
- Standard reporting mechanism
- TCL Directive commands
- Standard parameter defaults
- Allows text formatting using a standard formatter

TCL

- Script Language

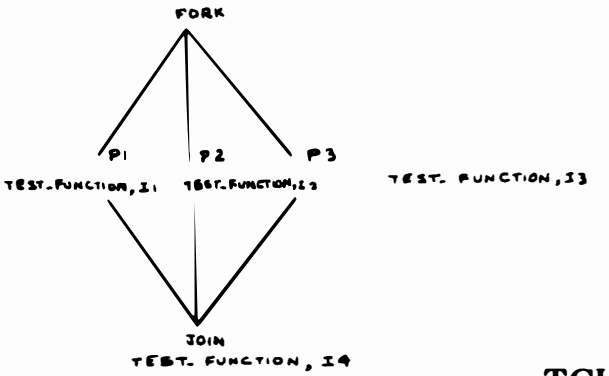
```
-- Compact Notation - make 10 procedure calls
Test_call, I1-110,P1 ({1-8.100,250},S[0,1])

-- Cross Product Notation - make 16 calls
Test_call,I*,P2 (100,S[0,{0-7},{0-1}]):C

-- Black Box Notation - makes 46 calls
Test_call,L*,P1 (S[1,0-1,{1..16382:20:R}])


-- Filling memory buffers
$INIT S1_Segment
20(5),'This is test data',10(0)
$END_INIT
```





## Semantics of remote procedure execution

- Fork and join semantics
  - Test_function,I1, P1 (1,2,3)
  - Test_function,I2, P2 (10,20,30)
  - Test_function,I3, P3 (5,6,7)
  - $end_context
  - Test_function,I4



## Syntax - Compact and Concise

```
Test_call,I1  -- Use all default parameters


-- or
$INIT ID    -- set up identifiers
I1= 10
I2 = 20
.
.
I36= 3600
$END
-- Perform 128 calls

Test_call,I*,P1 (100, S[I1-I2, I10-I17, I20-I21,
                        I30-I31, I35-I36 ])
```





## Expected Results

Results of a procedure call are verified by analysis of:

1. values of output parameters
2. errors

Example:

Check the output of the function call Test_call which returns the product of three numbers.

```
Test_call (1-5,1-5,1-5)
$EXPECTED_OUTPUT
RETURN_VALUE={1,8,27,64,125}
$END_OUTPUT

-- Negative test case - check error from two calls
Test_call (0,{1,-1})
$EXPECTED_OUTPUT
EXCEPTION = {ZERO,NEGATIVE}
$END_OUTPUT
```

## Standardized Reporting

$REPORT ALL -- details
$REPORT FAIL ONLY -- only failed details





## Directive Commands

| | |
|---|---|
| $RESET | return variables and context to original settings |
| $TIME | measure performance, individual and accumulated times |
| $REPORT | give detail results |
| $SETUP/UPSET | block logical groups of script logic |
| $EOS | End an interactive session. |
| $COMPARE | compare contents of memory buffers. |
| $INIT | set memory buffers, integers, strings, constants literal constants, to known initial values. |

## Standard Defaults

- Reduces number of Unknowns
- Simplifies Coding





## Leverage Xenix Tools

- SanScribe for Listing control

  table of contents
  page numbers
  headers/footers
- Xenix Standard I/O

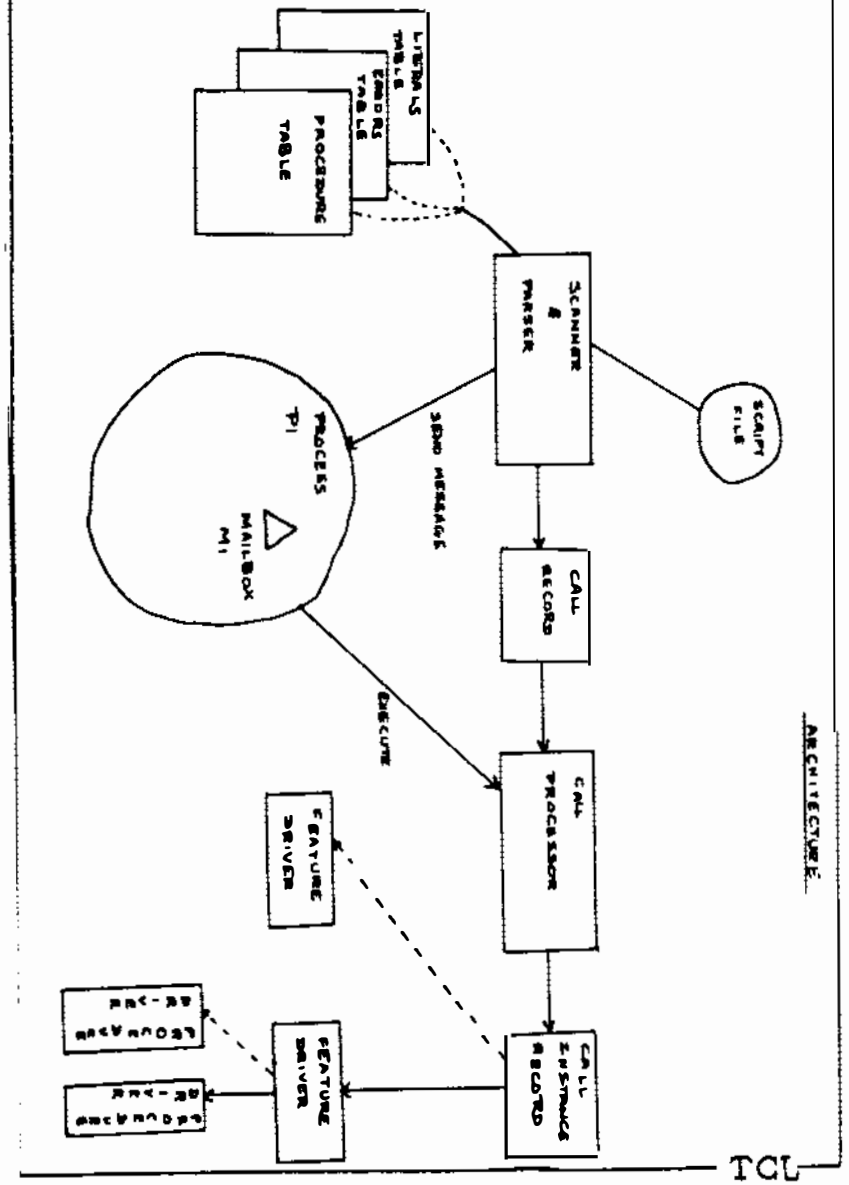
  source of input scripts
  terminal input
  output of detailed and summary files

## Design and implementation

- Implementation of TCI
- Building TCI for testing different programs







intel

## Scanner and Parser

- Tables for all procedures, errors and literals
- A procedure to look up an entry in any of the above tables
- Top down parser based on LL1 regular grammar
- Alphabet of the grammar

  - Integers, key_words, directive_commands, procedure names,
  - literals, error names, identifiers, strings, delimiters

- If there is no error in the command then the parser

  - Fills the CALL_RECORD with parameter values.
  - Manages remote procedure execution.
  - Calls the call_processor.

TCL





intel

## Call Processor

- Repeatedly generates each execution instance from the CALL_RECORD
- Fills in the CALL_INSTANCE_RECORD each time and calls the driver routine
- Uses an Abstract Data Type called set to store multiple values
- Uses standard routines for set manipulation

  - read(set), write(set), end_of_set(set),
  - create(set), reset(set), close(set)

- Example

  {1,2,3} --> parse --> create(set);write(set,1); write(set,2);..

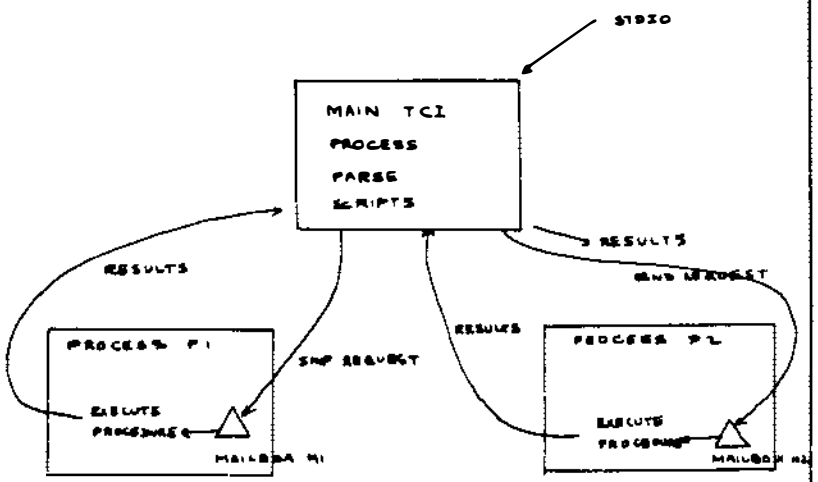
  Call processor --> read(set)

TCL

intel

## Contexts

- Internal Environment




TCL



intel

## Generating TCI for testing programs

- Driver procedures
  - A set of procedures, one for each procedure to
  - be tested
  - Create these driver procedures
- Tables to be filled
  - Fill the procedure table with the names of each
  - driver procedure and its entry point
  - Fill the error table with the names and values of
  - all the errors that can be returned by procedures
  - under test
  - Fill the literal table with the names and values
  - of all the symbols exported by procedures under
  - test

TCL

## Conclusions/Future Directions

- TCL dramatically reduces test development time
- TCL facilitates random testing
- TCL can be used for constructing unit tests
- TCL allows remote procedure execution
- TCL can be used for multi-processor testing

ABSTRACT

T is a new tool that automatically generates test case design and input data from requirements for software under test. Outputs from T include test case input data files containing ASCII data which can be reformatted and used as direct input to software being tested. Test cases generated by T will exercise every requirement for every one of the Most Probable Errors.

In order to use T software engineers must define requirements. After that, T will perform at the push of a button. Because there is so little effort involved in using T, test cases can be completed BEFORE design of the code begins. That means an engineer can "design out" a large percentage of the Most Probable Errors. The result is very high quality software.

Biography

Robert M. Poston
Programming Environments, Inc.
4043 State Highway 33
Tinton Falls, New Jersey 07753
(201) 918-0110

Robert M. Poston is president of Programming Environments, Inc., Tinton Falls, New Jersey. He has twenty years of experience managing and developing software for process control, simulation, radar, communication, business and operating systems.

From 1978 to 1982 he chaired the IEEE Software Engineering Standards Subcommittee. The first IEEE Software Engineering Standards Applications Workshop (SESAW) was initiated under his leadership. Mr. Poston served as Standards Coordinator between the IEEE Computer Society and the American National Standards Committee, X3, from 1982-84. In 1983 he was the IEEE Software Standards representative on the US People-to-People software delegation to China. He served as technical program chairman for Compstan 86. Presently, he is a member of the editorial board of IEEE Software magazine. His column on software standards appears regularly in that publication. Annually, he lectures at numerous professional symposiums and seminars throughout the world.

# T: The Automatic Test Case Data Generator

Now on the software scene is a new tool called T that automatically generates software test case design and input data. In this paper we will look at the industry needs that prompted the creation of this tool. We also will examine the concepts of software development which guided the evolution of T, and finally, we will take a close up look at T itself.

## TOP THREE DEVELOPMENT CONCERNS

Software developers face three major problems today: quality, cost and schedule. Notice the ordering of that list -- quality FIRST; cost important but secondary; and schedule last. If, as is normal on most projects, the order of those development issues is reversed, trouble is bound to follow.

Consider, for example, a product that is developed on time within projected budget and arrives first-to-market. But in the push to get to market, quality is compromised (overlooked). As soon as the end user gets hold of the product, development cost and schedule matters will fade away and quality concerns will come to the front to stay.

The user will ask

1. Does the product do what I need done? (functionality)
2. Does the product perform its functions fast enough? (performance)
3. Is the product easy to learn? (learnability)
4. Is the product easy to use? (usability)
5. How often does the product fail? (reliability)
6. How long does the product stay down when it breaks (or how long does it take to reload)? (availability)
7. How long does it take to get a "fix" or "workaround?" (maintainability)

Quality stays with a software product throughout its entire life cycle and is the primary concern of users today. A few years ago customers would buy an innovative, often inferior software product, because they wanted the latest in technology, and generally, high quality software was not available anyway. They would try to live with or work around problems. But now, with more and more competitive products entering the marketplace, buyers can be selective and current marketing statistics show people opting for quality over other considerations.

In the shaky world of the software industry offering a superior quality product means a chance to stay in business. A poor quality

software package can cost its producers a fortune in maintenance and can kill a product line. Remember VISI-ON? How about Easy Writer?

Development cost has to be a secondary consideration because even if development budgets are overrun, a product can still get to market with a chance to survive as long as quality has not been slighted. Perhaps the product will prove less profitable than desired because of unforeseen development costs, but if quality has been built in, sales usually follow and, most importantly, the reputation of the developer will be preserved.

Generally cost problems are better understood today than are quality issues. Software product life cycle cost statistics have been published widely. We now are acutely aware that over 30% of development dollars go to the activity most of us know as "testing." In fact, that activity in most organizations is actually two activities blurred together. First we have to find out if the product functions as specified (testing proper). Then if it does not, we have to perform rework or debugging (respecifying, redesigning, reprogramming and retesting).

We are beginning to find that rework adds significant, and usually unnecessary, expense to a project. Solid industry statistics on rework are not readily available, but information from my own informal polling in numerous organizations indicates that at least 25% of the software life cycle is devoted to rework.

There are several well-known ways to cut costs on a software project. The most commonly used ones are making the product smaller (descoping) and reducing rework.

Although schedule is relegated to third on our list, many of us know the pressure of trying to meet a tight development schedule, and certainly most managers put great importance on time tables. It is only when we can step back and look at schedule in the larger scheme of things that we see that it must take a back seat to quality and cost.

Schedule is a "right now" problem that has to be addressed frequently throughout a project, but often schedule problems sneak up on managers and developers alike. Did you ever notice how many projects appear to be "on schedule" right up to testing? In testing, however, all the failures that have been accumulating through earlier development phases begin to be discovered and then schedules are noticeably impacted. Indeed, testing the product looms up whenever we consider any of these three development problems: quality, cost or schedule.

The technical staff at Programming Environments, Inc. (PEI) of Tinton Falls, New Jersey has been working for several years to develop an approach to help minimize these problems. In seeking solutions we looked at the software development process with an eye to taking concrete measurements. After all, nothing in software can be made better until there is a reference point from which to improve. We found in (guess where) testing that quality characteristics could be

demonstrated. We could count the number of bugs and correlate that figure directly to rework costs and schedule. The development problems began to seem manageable once we could quantify them with the testing information. So we focused our approach on testing and visualized a software tool that would increase the effectiveness of testing and reduce the effort that goes into it. The realization of our concept is T.

Along the way to creating T we put a set of support elements in place, and now they are part of the T environment. Consequently, T is supported by a management policy (written description of what should be done in testing), an integrated set of test design techniques, standards for test documentation, testing metrics and a set of training seminars.

While there were some excellent books and publications available on hardware and software testing, as well as experts we could call on, most of the T project had to be guided by our own experience and common sense. Known techniques did not provide sufficient information for the automatic generation of test cases. We had to put together pieces of old knowledge in order to move on to new ground. We will take a brief look at some of the experience and concepts that influenced the development of T. Then we will turn our attention to the T product that resulted from our efforts and see how it is working today to help overcome the development problems.

## CONSIDERATIONS UNDERLYING T

Experts in hardware quality assurance have established that it is much better to prevent errors than to find them early. In software we are beginning to realize this, but most of our efforts are still oriented toward verification and validation or finding errors early. One of the requirements we put on T was that it must work to prevent the most probable software errors. That requirement affected T in all its life cycle phases.

Over 55% of failures discovered during testing are the result of mistakes or errors made in the first life cycle phase of defining the product's requirements according to Barry Boehm in his acclaimed book, Software Engineering Economics. IEEE Standard 830, Guide to Software Requirements Specifications, identifies and classifies probable requirements specification errors.

At PEI we used Standard 830 as a base on which to build a more detailed and complete list of most probable errors in requirements. Then we looked at all acknowledged requirements specification techniques (methodologies) in the industry (structured systems analysis, structured analysis and design techniques, etc.) to assess their success in preventing the errors on our list. None of the specification techniques were complete enough for our purposes, so we developed a set of rules of order for specifying software requirements. (see Figure 1)

These rules were very important to us for two reasons:

1. Each requirement had to be uniquely identifiable. That meant a requirement was easy to trace to test cases.

2. Every noun (data) used in a requirements statement had to be visible outside the unit under test. That meant test cases could be generated with externally visible information.

Test case design techniques amount to methods for selecting values for the input data (nouns) that will exercise the software under test. We found a number of techniques documented [1], but they were not put together into any coherent approach we could use in developing T. We employed our own Most Probable Errors (MPE) List (see Figure 2) to identify the values of inputs that had the highest probability of detecting failures. Then we selected test design techniques that would guide the developer to select the MPE values. Once the techniques were chosen and integrated into a test case generation procedure, we had a manual version of the PEI Testing Methodology -- the forerunner of T.

The manual PEI Testing Methodology was tried on a number of PEI projects and at numerous client sites. The most dramatic results were reported when the methodology was applied on a real-time, process control project called PIMS at Leeds & Northrup Company in North Wales, Pennsylvania. Mark Bruen of the Power Engineering section led the two and one-half year project to produce 27,719 lines of source code (not counting comments). One to three failures per 1,000 lines of source code is the average failure density factor being reported in professional journals today. [2] After rigorously following the PEI Testing Methodology the Mark Bruen team documented the remarkable failure density factor of .00057 per 1,000 lines of source code. The industry average productivity factor reported is four to ten delivered lines of source code per day. [3] The Bruen team showed 29 lines per staff day. After this success at Leeds & Northrup, we knew we were on the right track, and we pushed forward to complete the automation of T.

THE T TOOL

The currently available, commercial version of T is a software package that takes requirements information and separates it into a requirements list and five dictionaries: verb, event, condition, input data and output data. Partitioning this requirements information allows cross checking amongst the dictionaries for consistency, correctness and completeness. This is the starting point for automatic test case generation.

As discussed before two other components are unique to T: a knowledge base of the Most Probable Errors (faults) found in software today and a policy (algorithm) for the process of generating test cases. When the five dictionaries come together with the Most Probable Errors List and the policy for creating test cases, all the pieces that make T operate are in place.

What does T do exactly? T creates a file of data for each test case. The file contains ASCII data which is input to the software

under test. T generates test cases which will exercise each requirement for inputs with normal values and inputs with values derived from the Most Probable Errors List.

Because T is automated, a programmer can direct T to generate a set of test cases BEFORE design of the code begins. The set of test cases will exercise every requirement for every one of the most probable errors. If a programmer knows how code will be tested BEFORE design work begins, he or she is likely to design software to pass the tests and will avoid making errors in the first place. There goes rework.

Many programmers complain that it takes too much time and work away from the critical path to create test cases. They would rather spend time "making the code correct." T generates test cases for a programmer at the push of a button.

How about the quality aspects of T? How does T measure up when the user examines it?

1. Does T do what a programmer needs done? (functionality)

   - T captures requirements, verb, data, event, condition and descriptions.

   - T reports discrepancies in the five dictionaries.

   - T automatically generates test case design.

   - T automatically generates test case input data.

2. Does T perform its functions fast enough? (performance)

   - T generates an average of 30 test cases per minute. (The actual time a user experiences is dependent on the number of requirements statements and types of inputs to T.)

   - T captures data much faster than any human being can type.

   - T runs on a dedicated workstation (IBM PC-AT with 20 megabytes of hard disk, 1.2 megabytes of floppy disk and 640 kbytes of main memory). A user can schedule the running of T when the workstation is not is use (overnight).

3. Is T easy to learn? (learnability)

   - T is sold only to organizations whose managers, engineers, programmers and testers have been trained in the PEI Testing Methodology. This training is necessary to establish a comprehensive understanding of software testing according to PEI standards. It lasts three and

one-half days.

- T itself is easily learned by trained testers in one-half day.

- T control is menu driven for ease of learning.

- T data entry is prompted with definitions for ease of learning.

4. Is T easy to use? (usability)

   - T control is both menu driven (easy to use for the occasional tester) and command driven (easy to use for the dedicated tester).

   - T data entry can be accomplished in two ways: keyboard/screen form (easy to use for the occasional tester) and text stream with embedded key words (easy to use for the dedicated tester).

5. How often does T fail? (reliability)

   - T is being used to test T. We are presently experiencing a "mean time between failures" of two months. Updated results are being released periodically.

6. How long does T take to reload? (availability)

   - T comes with load T/unload T commands. Loading T requires less than four minutes on an IBM PC-AT.

7. How long does it take to get a "fix" or "workaround" for T? (maintainability)

   - To date we have not identified a failure in T that could not be repaired in two days by PEI staff. Technical assistance is available during weekday business hours.

Right now T is working in a Xenix/Informix environment for small collections of code (units of 10,000 lines of code or less). It also runs in a PC DOS + Informix + Brief environment. Work is progressing to expand T to the system level. This effort is being funded by the Department of Defense.

Figure 1 REQUIREMENTS RULES OF ORDER

1. USE A STANDARD FORMAT
   Reason: to lower the probability of "I do not know where to put [find] the information" errors

   Examples: DOD-STD-2167, DI-MCCR-80025, "Software Requirements Specification" ANSI/IEEE STD 830, 1984 "Guide to Software Requirements Specifications"

2. USE A CHECK LIST
   Reason: to lower the probability of "I forgot to consider that category of requirements" or incompleteness errors

   Example: ANSI/IEEE STD 830, 1984 "Guide to Software Requirements Specifications"

3. USE MODELS
   Reason: to lower the probability of errors of imprecision

   Examples: data structure models, data flow diagrams performance models

4. USE LISTS OF SENTENCES (avoid paragraphs)
   Reason: to lower the probability of omitting [losing or overlooking] a single requirement in designs or tests

5. USE SIMPLE SENTENCES (Non-compound verbs and nouns)
   Reason: to lower the probability of ambiguities in different interpretations of a sentence

6. USE END USER'S VOCABULARY (not programmer's)
   Reason: to lower the probability of the end user [the person spending the money] misinterpreting the requirements

7. USE ONLY NOUNS FROM A PROJECT CONTROLLED DATA DICTIONARY
   Reason: to lower the probability of making an error of inconsistency in defining data

8. USE ONLY VERBS FROM A PROJECT CONTROLLED FUNCTION DICTIONARY
   Reason: to lower the probability of making an error of inconsistency in defining software actions

9. USE ONLY NOUNS AND VERBS FOR OBJECTS AND ACTIONS VISIBLE EXTERNAL TO THE SYSTEM
   Reason: to lower the probability of restricting design trade-offs with requirements statements

Figure 2 GENERIC MPE CLASS LIST

| INPUT CLASS | For Timing Requirements | For a Table Data Structure |
|---|---|---|
| VALID | | |
| Low Boundary | slowest | minimun no. entries |
| Low Boundary+1 | slowest+1 | minimum no. entries+1 |
| Normal | average | average no. entries |
| Troublesome | crossover | first entry |
| High Boundary-1 | fastest-1 | maximum no. entries-1 |
| High Boundary | fastest | maximum no. entries |
| INVALID | | |
| Low Boundary-1 | slowest-1 | minimum no. entries-1 |
| High Boundary+1 | fastest+1 | maximum no. entries+1 |
| Abnormal | no input | no input |

---

1. Glenford Meyers, The Art of Software Testing, John Wiley and Sons, New York, 1979.

2. Nathan H. Petscenik, "Practical Priorities in System Testing," IEEE Software, Sept. 1985, p.18.

3. Barry W. Boehm, Software Engineering Economics, Prentice-Hall, 1981.

# Session 4

## METRICS

*"QA Metrics in the Workplace"*
Dave Dickmann, Hewlett Packard

*"Software Quality Measurement - A Methodology and its Validation"*
James L. Warthman, Computer Science Innovations, Inc.

*"A Comparison of Counting Methods for Software Science and Cyclomatic Complexity"*
Nancy Currans, Hewlett Packard

# QA METRICS IN THE WORKPLACE


## Abstract

Everyone is in favor of capturing and using software QA metrics to improve the quality of products. Translating theory from the textbook to the workplace to produce this improvement in a cost-effective manner is not, however, a trivial process.

This paper addresses that translation and use on one software project recently completed at Hewlett-Packard. In particular, the metric selection process, data capture, analysis and management use of the data is discussed.

The application of the data in real time to guide the development and test process for the project is followed. Actual project data is used to illustrate the discussion. Further use of the data to assist management in modifying the work environment and to assist the development engineers in modifying the development process is then discussed. The results from the next project undertaken by this development team are reviewed to determine the impact of the effort on long-term productivity.



Dave Dickmann
Hewlett-Packard Company
1000 N.E. Circle Blvd.
Corvallis, Oregon 97330


Dave is a software quality engineer in Hewlett-Packard's Portable Computer Division, supporting R & D in handheld calculators.

# QA METRICS IN THE WORKPLACE

Much has been written on the use of software QA metrics both to assist managers of on-going developments and to impact the way future developments are managed. This paper reviews the way metrics were actually used in the workplace on a recent Hewlett-Packard software development project.

As you can see from the first slide, this was a relatively small project, fitting comfortably into a 64K ROM. Because the product is used in unattended instrument control applications, product release criteria was set at the relatively tight limits shown.

The next slide shows the QA data collected and analysis done to determine when release criteria were met. The data we selected to collect and analyze was driven by three needs. Knowing when it was economically correct to release was foremost. Second, with nine modules to test, assigning efforts by types of testing and modules to concentrate on was important to avoid wasted effort. Finally, we are trying to improve our development process, so we wanted to know where defects were coming from.

One of HP's standard defect tracking systems was used to maintain defect details. A spreadsheet with associated graphics program was used to output the statistics weekly in both text and graphics form. Defect classification refers to the type of error made - incorrect flow, uninitialized variable, etc. One of the weekly summary reports is shown as an example in the next two slides.

What impact did this record keeping and reporting have on the project? A very heavy one as you can see in this comparison of what we thought we needed to do versus what we wound up actually doing. Part of the change can be attributed to code growth, but the majority was driven by our analysis of the QA data. This was particularly true in the code review area where re-reviews were scheduled when control flow errors were detected on initial review.

The analyzed data also guided the types of defects we looked for and in what modules/functions we devoted test hours to. The chart of defect sources shown here was updated weekly and led to some 20 changes in the functional test specification during review and test. Similarly, the chart of defects found versus hours of test effort applied modified test scheduling to direct more hours into areas where defects were being found.

The final "real time" use of the analyzed data was to determine when to move into final release efforts. Two methods were used. The first model shown, the Defect Discovery schedule, is used by determining (average) hours to find the most recent defect(s) and entering that in a linear display of hours per defect. The corresponding scale of defects is simply the current total number of defects divided by the marked percent of defects found. The second model, the Exponential Decay

Model, is really just an inverse of the Defect Discovery Schedule. We enter with defects found per hour of test and then best fit an exponential curve to the collection of data points. The area under the curve to the right of the current total test hours measures the defects remaining. Both models have been used extensively at HP and elsewhere and provide surprisingly accurate estimates of future problems.

This chart shows defects per 100 test hours versus calendar date, instead of defects per test hour versus total test hours. Since test hours per week were pretty constant during this project, the chart closely tracks with the Exponential Decay plot and shows the expected behavior. When our models indicated we were looking at 5-15 remaining defects we went into final release test mode - verify functionality one last time, hammer on defect fix areas, poll testers and designers for 'nervous' areas and verify completion of the functional test specification.

The last use of the analyzed data was to provide input to the next development, both for managers and for designers. By agreement with management, statistics maintained for individual designers were never shown to their supervisors - they were maintained by QA, discussed one on one with the individual designers involved, and destroyed after that.

What did it cost and how well did it work? The collection effort took about six engineering hours per week for 20 weeks, the analysis another three hours per week and report preparation about two hours per week. A total of 220 engineering hours were charged over the course of the project instead of the 130 hours that would have been expended on the minimum data collection and analysis required to track project release criteria.

The results seem well worth it. Defect detection rates exceeded the norm for similar projects and after over six months of sales only one field defect has been reported for the product. The design team is well into its next project, posting small red pennants at their desks when they do not want to be interrupted. They report less stress and more output than on past projects. Code reviews are well along and are showing a much lower incidence of coding errors. Management, the design team and QA all consider the effort to have paid back its investment many times over. Similar collection and analysis efforts are underway on virtually all projects in this lab now with full expectation of getting more-than-adequate payback. Software QA metrics do have a place in the workplace and can contribute to the bottom line.

# PROJECT OVERVIEW

- **ROM Based Application Program**
- **4 Main Modules, 3 Minor Modules, 2 Utilities**
- **7 Klines Uncommented Source Code**
  - 4.5 Klines BASIC
  - 2.5 Klines Assembler
- **Size Not Critical**
- **Execution Speed Important**
- **Release Criteria**
  - No open known defects
  - Not over 3 trivial defects last 150 hrs test
  - Estimated remaining defects:
    0 Fatal
    1-3 Normal
    0-10 Total

# QA DATA

## Data Collection

By Module
KLines Uncommented Source Code
Hours Expended
Type of Activity
Details of Defects Found
Functional Tests Completed

## Data Analysis

Defect Severity
Defect Classification
Activity When Found
Earliest Activity Defect Findable

## Statistics Maintained

By Module and Overall
By Severity
Updated Weekly
Defects/100 Hours
Defects/KLine

# 3/20 QA REPORT

**WEEK 8/05 (193 TEST-HR)**

| MODULE | DEFECTS |
|---|---|
| M1 | 0/0/4/0 = 4 |
| M2 | 0/0/0/0 = 0 |
| M3 | 0/0/1/3 = 4 |
| M4 | 0/0/0/1 = 1 |
| m1 | 0/0/0/1 = 1 |
| m2 | 0/0/0/1 = 1 |
| m3 | 0/0/0/0 = 0 |

**WEEK 8/12 (177 TEST-HR)**

| MODULE | DEFECTS |
|---|---|
| M1 | 0/0/1/2 = 3 |
| M2 | 0/0/0/0 = 0 |
| M3 | 0/0/0/1 = 1 |
| M4 | 0/0/1/2 = 3 |
| m1 | 0/0/0/1 = 1 |
| m2 | 0/0/0/1 = 1 |
| m3 | 0/0/0/0 = 0 |

# 3/20 QA REPORT

- **CONCERNS**
  - M1, M3, M4
  - Retest Basic Areas
  - Specialized Testing
    - Retest all August Defect Fixes
    - Test in Related Areas
  - Function Coverage

- **NEW RELEASES**
  - 8/20 Release Available
  - 8/23 Friday "Final" Version
    - Begin Using Friday AM

- **CODE RELEASE TO MASK**
  - Thursday, 8/29
    - Risk Assessment Meeting
  - Release Criteria
    - 150 Hrs. 8/19-8/22
      - 3-4 Defects, No Fatal
    - 150 Hrs 8/23-8/29
      - 2-3 Defects, Only Trivial

# SOFTWARE QA EFFORT

| TASK | INITIAL SCHEDULE | ACTUAL |
|---|---|---|
| Design Reviews | | |
| Sessions | 10 | 4 |
| Eng-Hours | 60 | 140 |
| Code Reviews | | |
| Sessions | 15 | 32 |
| Eng-Hours | 90 | 265 |
| Functional Testing, Hours | 900 | 1890 |
| Klines Source Code | 4.5 | 7.0 |
| QA Eng-Months/Kline | 1.3 | 1.8 |

# DEFECT SOURCES

| SOURCE | WHEN FOUND: DESIGN REVIEW | CODE REVIEW | FUNCTNL TEST | TOTALS |
|---|---|---|---|---|
| Incorrect Control Flow | | 47 | 23 | 70 |
| Incorrect/Ambiguous Prompt/Msgs | 21 | 10 | 16 | 47 |
| Uninitialized/Wrong Variable Usage | | 20 | 21 | 41 |
| Error Handling | | 26 | 1 | 27 |
| Improperly Coded Algorithm | | 13 | 8 | 21 |
| IF/THEN Predicate Wrong | | 10 | 6 | 16 |
| Major Enhancement Defects | | | 12 | 12 |
| Incorrect Flag Usage | | | 8 | 8 |
| Boundary Value Failure | | | 7 | 7 |
| All Others | 3 | 6 | 18 | 27 |

# FUNCTIONAL TEST

1890 HOURS

| DEFECT SEVERITY | REGRESSION 0(100%) | 1(80%) | 1(70%) | 1(40%) | TOTALS |
|---|---|---|---|---|---|
| CRITICAL | 7 | 0 | 0 | 0 | 7 |
| SERIOUS | 16 | 0 | 0 | 0 | 16 |
| NORMAL | 24 | 7 | 4 | 0 | 35 |
| TRIVIAL | 27 | 17 | 11 | 7 | 62 |
| TOTAL | 74 | 24 | 15 | 7 | 120 |

| MODULE | HOURS | DEFECTS | MODULE | HOURS | DEFECTS |
|---|---|---|---|---|---|
| M1 | 410 | 27 | m1 | 180 | 13 |
| M2 | 400 | 22 | m2 | 180 | 6 |
| M3 | 330 | 19 | m3 | 20 | 1 |
| M4 | 370 | 20 | U1 | * | 7 |
| | | | U2 | * | 5 |

* No testing unique to these modules. They were tested through extensive use while testing other modules.

# FUNCTIONAL TEST

1890 HOURS

| | | |
|---|---|---|
| Defect Detection Rate | Start | 6.0/100hrs |
| | Peak | 19.2 (@35%Pt) |
| | End | <1.0 |

Detection Rate Decayed Exponentially from 690 hrs to end.

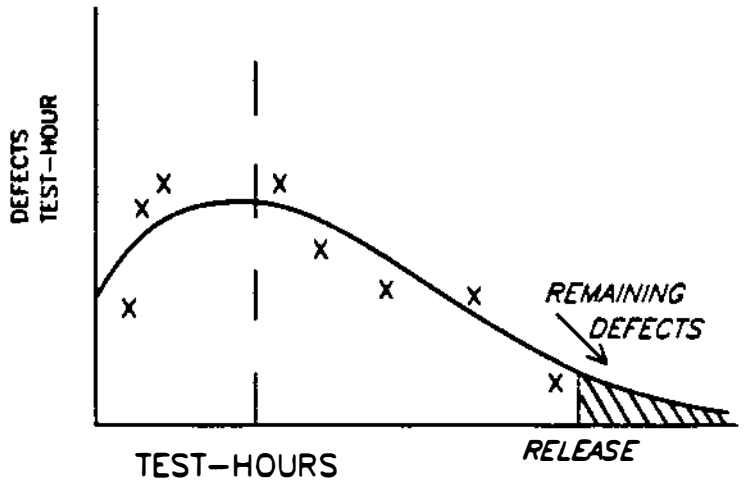
# DEFECT ESTIMATING

## Defect Discovery Schedule

| 25% of defects found at | ≤ 2 hrs/defect rate |
|---|---|
| 50 | 5 |
| 20 | 10 |
| 4 | 20 |
| 1 | ≥ 50 |

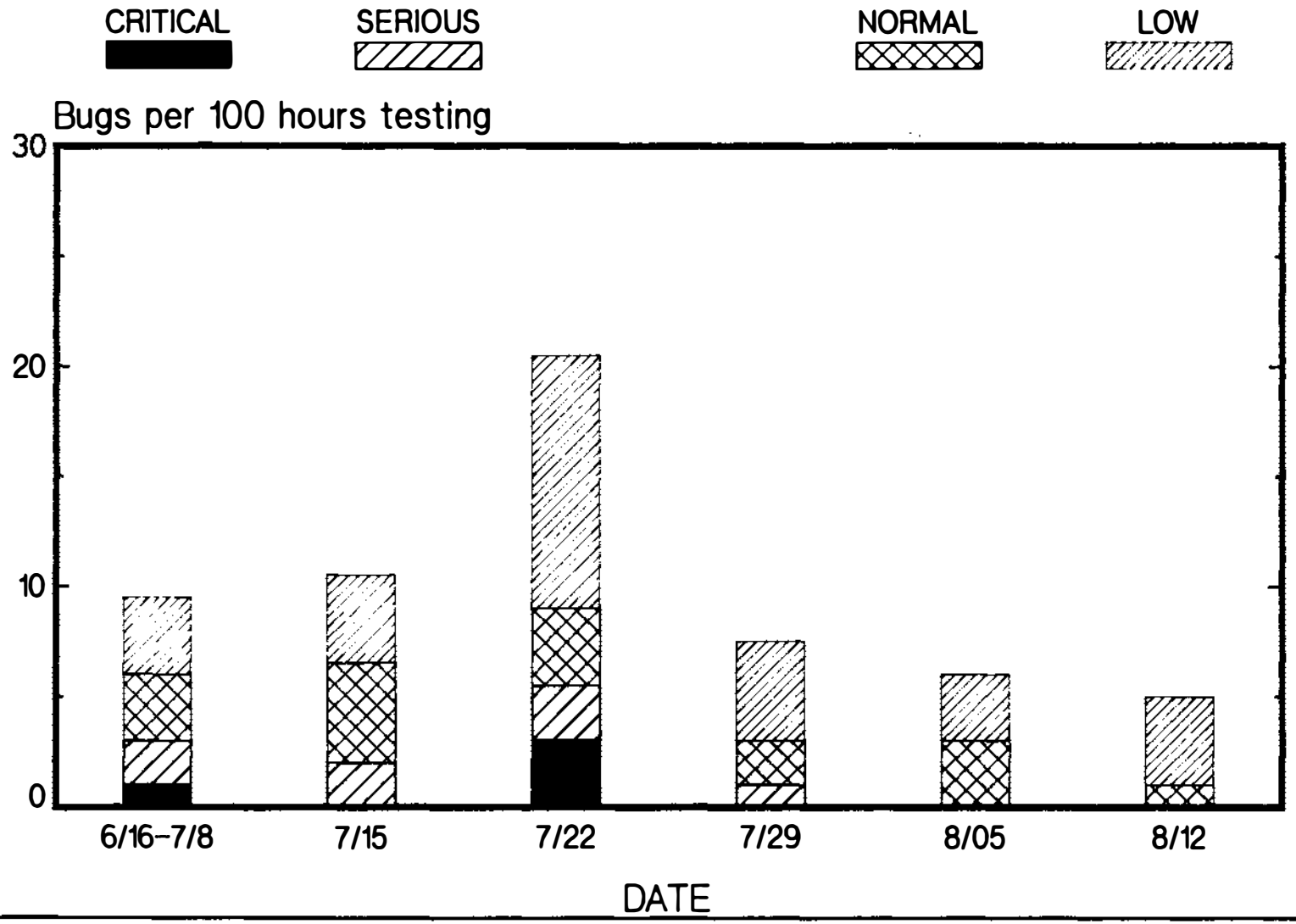
# DEFECT ESTIMATING

## Exponential Decay Model



# DEFECTS DETECTED PER 100 HOURS

## June 16 - August 12






HEWLETT PACKARD

## LESSONS LEARNED

### DISCUSSED IN OPEN FORUM

This was done in designer non-specific discussion.

- Working Environment (Interruptions Affected Quality)
- Manual Preparation (Underestimated)
- Multiple Technical Users Consulted About Design
- Peripheral Equipment Shortcomings
- Code Commonality Not Exploited

## LESSONS LEARNED

### OVER COFFEE WITH DESIGNERS

Candid discussion with each designer to give feedback on bug rates, problem areas they could improve, etc.

This was non-threatening and received as positive education leading to personal improvements. They knew this data would not be shared with their manager.

# SOFTWARE QUALITY MEASUREMENT – A METHODOLOGY AND ITS VALIDATION

**Presented to the**
**Fourth Annual Pacific Northwest Software Quality Conference**

James L. Warthman
Computer Science Innovations, Inc.
1280 Clearmont St, NE Palm Bay, FL 32905-4094
(305) 676-2923

## ABSTRACT

This paper describes the preliminary results of a Software Quality Measurement Demonstration (SQMD) study which was performed for the U.S. Air Force Rome Air Development Center (RADC). The goals of this study were to evaluate RADC's software quality framework and the methodology for its use, and recommend improvements to RADC. This paper describes the elements of the framework and explains how they are used. It gives an overview of the study followed by a summary of the results. This paper contains the opinions of the author which are not necessarily the same as the views of RADC.

## BIOGRAPHY

Mr. Warthman has been with Computer Science Innovations, Inc. for over three years and is currently program manager for the Software Quality Measurement Demonstration project. He recently presented a paper on software quality specification to the Ninth Minnowbrook Workshop. He has participated in standards activities, including two ANSI committees. Prior to joining CSI he held software engineering positions with Harris Corporation and the U.S. Air Force. Mr. Warthman is a member of the IEEE Computer Society.

## I. INTRODUCTION

This paper describes the preliminary results of a Software Quality Measurement Demonstration (SQMD) study performed for Rome Air Development Center (RADC) by Computer Science Innovations, Inc. (CSI). The goals of this eighteen month effort were to evaluate RADC's software quality measurement framework and to validate selected metrics for adequacy and correctness. Conclusions were drawn regarding the utility of the methodology as a quantitative input to the overall quality assurance process, and specific recommendations were made to improve the methodology and the metrics [1,2].

## II. RADC SOFTWARE QUALITY FRAMEWORK AND METHODOLOGY

Software Quality Assurance was, at one time, primarily an exercise of testing to insure that the product satisfied the requirements. It is now generally acknowledged that this type of 'QA' is inadequate because quality must be built in, not tested in. The traditional approach typically resulted in systems that were expensive to modify and maintain, and often did not meet the needs of operational personnel.

### A. History

The need for a quantitative way to specify and measure software quality was recognized and, in 1976, RADC began the study of a software quality framework. At that time, an effort was undertaken to explore methods of specifying and measuring software quality in a quantifiable manner. An important goal was to make this technology available to the Air Force personnel who contract for the development of software - the acquisition managers. This study defined software quality in a hierarchical fashion. At the top of this framework were eleven quality factors which represented user-oriented views of quality such as *reusability*, *maintainability*, *reliability*, and *correctness*. Each factor was described in terms of software-oriented attributes called criteria. Examples of criteria are *traceability*, *consistency*, *completeness*, *modularity*, and *machine independence*. Each criterion was further defined in terms of metrics which were quantitative measures of an attribute. Metrics were items such as quantity of comments, completeness checklist, and complexity measure. This framework would then be used to specify, measure, and predict the quality of a software system at various points in its life cycle. This initial effort resulted in publication of a Preliminary Handbook on Software Quality for an Acquisition Manager [3].

### B. Framework

Subsequent studies were performed to refine and enhance the framework. Quality factors were added, criteria definitions were improved, and metrics were added, modified, and deleted as needed. The resulting framework contains thirteen quality factors, twenty-nine criteria, and over three hundred individual metric questions [4,5].

Factors are grouped into three areas or acquisition concerns: performance, design, and adaptation. Performance factors, such as *efficiency* and *reliability*, are concerned with the correct, error free functioning of the software. Design factors, such as *maintainability*, are concerned with the extent to which the software is verifiably correct and easily maintained. Adaptation factors, such as *portability* and *reusability*, are concerned with the effort necessary to change the software to meet new requirements.

Each factor is defined by one or more criteria. There is overlap such that a single criterion may contribute to the definition of multiple factors. For example the criterion *visibility*, which has to do with software testing, is used by the factors *maintainability* and *verifiability*. Thus, a development effort which provides for good *visibility* in the form of thorough testing should result in highly verifiable and maintainable software.

Criteria, like factors, are defined by lower level attributes. Criteria are measured by metrics which are quantifiable measures of specific software characteristics. These metrics consist of one or more questions, called metric elements. Answers to the questions are used to calculate a value between 0 and 1 for each metric, 1 being the best possible score. Each metric could be classified as either binary (existence of a property) or as a relative quantity (e.g., ratio of the number of lines of comments to total lines of code).

C. Methodology

The RADC methodology can be viewed in two parts. First, the desired levels of quality are specified for each of the thirteen quality factors. As development of the software proceeds, documentation and code is evaluated to determine the degree of achieved quality. These two steps are referred to as quality specification and quality evaluation.

Quality specification is intended to be performed by the software acquisition manager (SAM). He first determines which of thirteen quality factors are applicable. He specifies factors separately for each major software function.

Next, he develops an initial set of quality goals for these factors. Goals, at this stage, are expressed as desired levels of quality, i.e. excellent, good, or average. The SAM must take much into consideration when he sets these initial goals, including system level quality concerns, user concerns, and quality issues that are typical of the type of system which is under development.

After the SAM has established initial factor goals, he examines them for technical feasibility . He looks for potential problems which might arise if strong adverse relationships exist among these goals. It may be necessary to alter the initial quality goals if he determines that they represent a high technical risk. For example, an excellent rating for the factor *efficiency* may not be feasible if an excellent rating is also desired for the factors *reliability* and *expandability*. This is because some characteristics of efficient software are in conflict with characteristics of highly reliable and expandable software. The SAM makes adjustments to the initial quality factor goals as necessary, based on the technical feasibility evaluation.

Next, the SAM studies the cost implications of the quality goals. He estimates the additional costs of achieving the specified quality goals, and also the potential returns over the life cycle of the software. In this way, he can make judgements about the cost-effectiveness of the specified quality goals. If his study of cost implications show that the initial quality goals are not cost effective, the SAM modifies these initial goals.

The SAM expresses the final quality factor goals numerically on a scale from zero to one. A goal of excellent typically translates into a range of .90 - 1. Good is represented by a range of .80 - .89, and average is represented from .70 - .79. These numeric factor goals are incorporated into the software specification as quantitative software quality requirements.

After choosing quality factor goals, the SAM assigns weightings to the software criteria which make up each factor. In this way, he can emphasize or de-emphasize individual criteria. For example, *efficiency* may be an important factor for a software system. However, this system might not communicate with external systems. In such a situation, the criterion *communications*

*effectiveness* would be de-emphasized, and the other criteria which make up *efficiency* would receive additional emphasis.

Finally, the SAM selects the metrics which will be used to measure the achieved quality. He may eliminate individual metrics and metric elements which are not applicable to this software system, or which require an inordinate amount of effort to measure.

Quality evaluation is the process of measuring software development products to determine achieved quality. DOD-STD 2167 [6] defines the software life cycle and the development products that are used by the methodology. Twenty nine separate document types are evaluated, such as Software Development Plan, System/Segment Specification, Software Requirements Specification, Software Detailed Design Document, Source Code Listings, Software Test Procedure, and Software Test Reports. The developer or independent validation and verification (IV&V) contractor applies worksheets to measure aspects of these documents near the end of each life cycle phase. Each worksheet pertains to one life cycle phase. It consists of a series of questions, whose answers are expressed numerically as a decimal between 0 and 1, inclusive. In addition to the numerical answers which are used to calculate metric scores, the rationale for the answers is captured for later analysis.

Next, the answers to worksheet questions are used to calculate scores for each criterion. Criteria scores are, in turn, used to calculate scores for each applicable quality factor using the criteria weighting mentioned above. These calculations are performed with the assistance of scoresheets which help structure the necessary information for the calculations. A separate set of quality factor scores is calculated for each software function.

Factor scores are then analyzed, and compared against the original quality factor goals. Scoring trends are examined over several life cycle phases. Significant variations from the original factor goals are analyzed to determine the cause. Finally, actions are recommended to correct these variations. Results are thoroughly documented in a Software Quality Evaluation Report.

D. Software Quality Guidebooks

RADC's methodology is documented in a three volume document, Specification of Software Quality Attributes [7]. These volumes, referred to as the guidebooks, were produced by Boeing Aerospace Company under contract to RADC. The first volume comprises Boeing's final technical report, and is not required for implementing the methodology.

The second volume, called the Software Quality Specification Guidebook, describes the framework and the portions of the methodology which would normally be applied by the SAM. Also referred to as guidebook volume II, it includes the procedures for specifying software quality and assessing compliance with quality requirements.

The third volume, called the Software Quality Evaluation Guidebook, also describes the framework. Referred to as guidebook volume III, this volume describes the portions of the methodology which would normally be applied by the software developer or an IV&V contractor. Procedures for evaluating quality and for reporting on the results are defined.

## III. EVALUATION

The purpose of our study was to evaluate RADC's methodology and validate the framework components which have been developed and refined over the past ten years. In particular, RADC was interested in recommendations for further improving the methodology. This was

accomplished by applying the methodology to a test project, collecting information about the process, and recommending improvements. This section describes the test project and our application of the methodology. Important deviations from RADC's methodology are noted.

A. Senior Battle Staff Decision Aids (SBSDA)

The SBSDA system consists of four prototype decision aids [8,9]. These aids were developed to demonstrate that Artificial Intelligence, Decision Analysis, and Operations Research techniques can assist the Tactical Air Force Senior Battle Staff decision makers. Our study involved two of the decision aids, the Resource Apportionment Aid (RAA) and the Enemy Performance Assessment Aid (EPAA). The primary use of the RAA is to recommend the most efficient use of air power and provide the rationale for this apportionment. The EPAA is intended to assess the current state of an enemy's combat capability.

RADC sponsored a parallel study using the other two decision aids. The objective was to obtain a second, independent evaluation of the framework and methodology, and corresponding recommendations.

B. Quality Specification

Software quality specification was performed in accordance with the instructions in the Software Quality Specification Guidebook, Volume II. The following paragraphs summarize the major procedures. More detailed information may be found in [7].

First, the major software functions were identified, and initial quality goals were assigned for each function. These goals were expressed in terms of the importance of each quality factor. That is, each factor was described as excellent, good, average, or not applicable. Initial goals were derived from a study of the SBSDA documentation and typical quality concerns for command and control software, according to the guidebook. In addition, quality requirements surveys were used to gather inputs from the SBSDA developers and the SBSDA acquisition manager, since it was expected that they would have strong opinions on the importance of each factor.

After the initial quality goals were chosen they were examined for technical feasibility. This procedure involved consideration of beneficial and adverse relationships among factors. The ability to attain high quality for a factor may be impaired if high quality is also desired for another, conflicting factor. Similarly, certain combinations of factors have positive interrelationships, thus reducing the technical difficulty in attaining the desired quality goals. When the technical feasibility of attaining a certain set of goals was determined to be low, adjustments were made to the quality goals, in effect raising the feasibility.

The next activity involved considering the cost ramifications of the quality factor goals. In many cases, extra effort is required to achieve a high level of quality. However, in theory lower costs during the operations and maintenance (O&M) phase more than compensate for higher development costs. This step examines such relationships in detail. The guidebook indicates that this step is not needed when performing quality specification outside the system life cycle context. Therefore, the cost effects were not studied for the SBSDA.

Next, the relative importance of the criteria was examined. Each factor is defined in terms of one or more criteria, which are not necessarily equally important to the factor. Criteria weighting allows some criteria to be emphasized and others to be de-emphasized or eliminated. Weighting matrices were developed to define the contribution of each criterion to each quality factor.

The last activity in quality specification, according to the guidebook, is to select individual metrics and metric elements which will be used to measure the quality during the development process. Effectively, this involved the elimination of any metrics which did not apply to the system being measured. In the case of SBSDA, no metrics were eliminated, since it was an objective of the study to examine all metrics.

At the completion of quality specification, the guidebook assumes that quality factor goals will be included in a System/Segment Specification. Since the SBSDA development was already completed, the quality goals were documented in a Software Quality Requirements Report (SQRR). This document included intermediate findings and rationale, as well as the final quality goals.

C. Quality Evaluation

Software quality evaluation was performed in accordance with the instructions in the Software Quality Evaluation Guidebook, Volume III. The following paragraphs summarize the major procedures. More detailed information may be found in [7].

Evaluating the quality of software involved answering questions which are relevant to the current development phase. These questions are contained on worksheets which are tailored to each life cycle phase, as defined by DOD-STD-2167 and the associated data item descriptions.

Worksheet questions are oriented to the products of a 'standard' development. Since the SBSDA documentation didn't follow these standards the process of answering worksheet questions was very difficult and time consuming. In many cases, information needed to answer the questions simply could not be found.

After answering all questions on a particular worksheet, the criteria and factor scores for that worksheet were calculated. This process was aided by a series of scoresheets in the guidebook. The scoring process was repeated for each worksheet, and resulting scores were graphed for each quality factor. These scores were then compared against the original quality factor goals to assess compliance with the goals. The guidebook suggests that corrective actions be recommended which address quality deficiencies. Since the quality goals were established after the completion of the SBSDA development, no corrective actions were possible. Therefore, we made no attempt to provide such recommendations.

## IV. SUMMARY & CONCLUSIONS

Our application of RADC's software quality methodology to two test systems has resulted in a number of important findings and recommendations to RADC. Although improvements are needed in both the framework and the guidebooks, significant benefits may be obtained by selectively applying the methodology as it presently exists. The following paragraphs outline our major findings and recommendations.

A. Framework

Using the framework helps, in general, to raise awareness of quality concerns. Two areas which have traditionally been overlooked are the quality needs of the end-user, and software O&M considerations. Quality factors such as *usability* and *survivability* allow end-user performance requirements to be taken into account. The framework also allows the SAM to consider O&M ramifications early in the development. Factors such as *expandability*, *flexibility*, *portability*, and *reusability* focus on the ability of the software to be adapted to changing requirements. By

addressing these areas early in the development process, the resulting software systems can be expected to better meet the user's requirements, while being more cost effective to maintain.

Although the framework allows the SAM to address quality issues, it hasn't been updated to keep pace with technology. It is imperative that quantitative measurements of software quality reflect the use of new methodologies, computer architectures, programming languages, etc. Clearly, the use of new tools can have a profound effect on many aspects of software quality.

When it was first formalized, the framework was intended to evolve with technology. It was expected that new metrics and criteria would be added to measure the use of new software engineering practices. Examples of technologies which are not measured by the metrics include the use of knowledge based systems, fault tolerant systems and fourth generation languages. Through both current and planned studies, RADC will likely enhance the framework to more completely address new software technologies, as military use of these technologies becomes more widespread.

Certain framework elements need clarification or refinement. As an example, consider the criterion *modularity*. This criterion is an indication of the degree of coupling and cohesion between software units. When applied to the design or the code, this measure is a valuable indication of the use of structured techniques. However, coupling and cohesion are currently measured very early, during the Software Requirements Analysis phase. Since coupling and cohesion have not been shown to have meaning with respect to higher levels of abstraction such as the functional decomposition of a system, these metrics of *modularity* should not be measured until preliminary design. Instead, during the Software Requirements Analysis phase, *modularity* should measure the presence of design and coding standards which should result in the production of highly modular code.

Several other factors, criteria, and metrics were found to need revision. For the most part, these problems with the framework reflect a lack of measurements which are important to a particular factor or criterion at a certain life cycle phase.

DOD-STD-2167 allows for the specification of software quality goals. To accomplish this, the SAM needs guidance in the quality specification process. The framework and guidebook volume II provide good guidance for specifying quality in accordance with this new standard.

B. Guidebooks

The guidebooks document the methodology for specifying and evaluating software quality. In addition, they contain the rationale for performing certain procedures, a history of the quality framework, and examples of its use. This tends to cause confusion when trying to follow the instructions for applying the methodology. Little differentiation exists in the guidebooks between the theory, examples, and actual procedural steps which are to be followed. Also, in several areas more detailed information is needed. For example, it is unclear what steps are required to score the quality factors after completing the worksheet questions.

Our recommendation is to restructure the guidebooks. Examples, theory, and history could be removed from volumes II and III, and published as a Software Quality Measurement Text. Procedures for quality specification and evaluation could then be re-written to expand the detail. These actions would greatly improve the usability of the methodology.

The quality specification process suffers from a high degree of subjectivity. Quality goals for each factor are expressed in terms of excellent (E), good (G), average (A), or not applicable (N/A), and these terms can mean very different things to different people. This problem is especially evident

with the quality requirements survey. Prospective users and maintainers of the software system are asked to provide a goal (E, G, A, or N/A) for each factor. Responses to this survey during our study demonstrated that individuals can have very different interpretations, both of the rating scale, and of the factor definitions themselves.

Our solution is to define a new set of 'acquisition criteria' and 'acquisition metrics' for each factor. These new criteria and metrics would form a hierarchy that is parallel to the present software-oriented criteria and metrics. Acquisition criteria and metrics, as the name implies, would characterize each factor in terms of detailed acquisition concerns. The questions could be answered, in most cases, very objectively.

Consider, for example, the factor *integrity*. The current methodology requires that a goal of E, G, A, or N/A be selected. It is difficult to differentiate between excellent and good levels of *integrity*. The alternative would describe *integrity* in terms of questions that relate to this factor. Typical questions might be: what is the highest classification of information in the system? Will multiple levels of classification be supported? Will communications interfaces be used? The answers to these questions would be quantified, and used to calculate a quality goal for *integrity*. This acquisition hierarchy would greatly reduce the subjectivity involved in establishing initial quality goals. It would shorten the learning curve associated with learning the factor definitions. The answers to the questions would also be useful in resolving technical feasibility conflicts among the factors.

An important strength of the methodology is the extent to which it supports an in-depth study of software quality issues. It is important, not just to specify the desired quality goals, but also to consider the interrelationships among the factors. The guidebooks provide the mechanisms to examine both technical and cost ramifications of the quality specification. With this capability, the SAM can study various options, and make informed choices to arrive at a cost effective set of quality requirements.

The ability to specify quality factors such as *efficiency*, *survivability*, and *usability* has one potential pitfall which must be considered. That is, care must be taken during the acquisition process that quality factors do not become a substitute for robust functional and performance requirements. To lessen this possibility, we believe the guidebooks should discuss the relationship between quality, functional, and performance requirements. It should be made clear that quality specification is intended to support rather than supplant complete requirements in the other areas.

For several reasons, worksheet questions were difficult to answer. The information necessary to answer a question was sometimes difficult or impossible to locate. This led to situations in which it was unclear if 'no' or 'N/A' was the appropriate answer. This choice can have a profound impact on the quality scores, since N/A questions are effectively eliminated from scoring, while 'no' answers reduce the overall score. In some cases, all metrics for an entire criterion scored 'N/A'. The guidebooks are silent about how to accommodate such a condition.

Several actions could be taken to reduce the effects of these problems. First, the worksheet questions could be organized differently. Presently they are listed alphabetically by metric element identifier. A much more useful order would be by the document and paragraph where the answer should be found. For example, all questions whose answers should be found in the Software Development Plan could be listed together.

We have recommended that the ambiguity concerning 'N/A' answers be eliminated by removing that choice from all questions. That way, any time an answer to a question could not be found, the answer would default to 'no'. It would be the responsibility of the customer (normally the government) to eliminate all questions which are not applicable. Of course, the developer could request that specific questions be designated N/A, but that choice would rest with the customer, not the developer.

Guidebook volume III describes analysis procedures which are to be followed after scoring each worksheet. This involves looking for trends in factor scores from one worksheet to the next. It also involves comparing measured factor scores with specified factor goals. Variations must be analyzed and corrective actions recommended. This process would be valuable, if scores on each worksheet and specified factor goals were normalized, but they are not. Therefore, it could be misleading to compare scores in the way the guidebook suggests. Without a statistical base, no conclusions can be drawn when, for example, a factor goal of .8 is specified and .76 is measured. Likewise, no conclusions can be drawn when a factor measures .76 after preliminary design and .56 after detailed design. Although quality requirements can be specified, there is no way to verify that they have been achieved.

It is RADC's intent to develop a strong statistical basis for normalizing these measurements. Until this has been accomplished, however, the guidebooks should be modified to explain problems in attempting to compare scores, rather than encouraging this process.

Building a statistical basis to use in normalizing the factor scores presents another problem. If the framework evolves with technology, as it should, then attempting to develop normalization functions will be rather like trying to hit a moving target. It will be difficult, at best, to build the necessary statistical base when metric, criteria, and factor definitions are changing.

Possibly the most important benefit of the methodology is in the area of problem detection. Although it is not yet feasible to compare measured quality with specified quality, it is practical to focus on factors and criteria with very low scores. This can serve as a trigger that something is amiss, and individual metrics can be examined to uncover specific weaknesses. Since this can be done after each life cycle phase, it is possible to take corrective action quickly, thus minimizing the cost to solve problems.

C. Future Directions

RADC's framework is a useful tool in its present form. To make appropriate use of the methodology, it is important to remain aware of the limitations, particularly in the area of interpreting factor scores. Keeping that in mind, the application of the framework and the guidebooks presents important benefits to software acquisition managers and developers.

To further enhance its utility, RADC continues to sponsor studies to refine the framework and improve the utility of the methodology. The framework must evolve to account for new technology and improved software engineering practices. Additional metrics are needed, and in some cases existing criteria and metrics need to be re-structured.

The guidebooks must be revised, both as the framework evolves, and to improve the usability of the methodology. More examples are needed, and the procedures for performing quality specification and quality evaluation must be made more 'cookbook-like'.

Various organizations, both in government and industry, are applying the RADC methodology to software development efforts. These groups can provide important feedback to RADC which will aid in the further refinement of the framework and guidebooks.

Efforts have been ongoing to automate certain portions of the methodology, notably quality evaluation, factor scoring, and analysis. Many metrics lend themselves to automation, particularly those used during detailed design and coding phases. Tools which automate metric collection should be made widely available. Continued emphasis on tools is important, and studies should be conducted to automate more of the methodology.

Software continues to account for a progressively greater share of every system development dollar, and the funds spent to operate and maintain software greatly outweigh those spent on software development. To make the best use of the software dollar, it is important that quality be given appropriate attention, particularly early in the life cycle, where the potential benefits are the largest. The RADC software quality framework, if used correctly, provides the mechanism for that much-needed attention.

## ACKNOWLEDGEMENT

I wish to thank Mr. Tod Loebel and Ms. Carolyn Midwood for their contributions to this study. Both worked long hours, implementing the methodology and reporting significant findings.

## REFERENCES

[1] *Software Quality Measurements Demonstrations*, Statement of Work, Rome Air Development Center, PR No. B-5-3253, contract no. F30602-85-C-0180, October 15, 1984

[2] Warthman, J.L., *Software Quality Measurement Demonstration Results*, Interim Technical Report to RADC for contract no. F30602-85-C-0180, August 15, 1986

[3] McCall, J.A., Richards, P.K., and Walters, G.F., *Factors in Software Quality*, Report No. RADC-TR-77-369, Rome Air Development Center, Griffiss Air Force Base, NY, November, 1977

[4] McCall, J.A., and Matsumoto, M.T., *Software Quality Metrics Enhancements*, Report No. RADC-TR-80-109, Rome Air Development Center, Griffiss Air Force Base, NY, April, 1980

[5] Bowen, T.P, et.al., *Software Quality Measurement for Distributed Systems*, Report No. RADC-TR-83-175, Rome Air Development Center, Griffiss Air Force Base, NY, July, 1983

[6] *Defense System Software Development*, DOD-STD-2167, June 4, 1985

[7] Bowen, T.P., Wigle, G.B., and Tsai, J.T., *Specification of Software Quality Attributes*, Report No. RADC-TR-85-37, Rome Air Development Center, Griffiss Air Force Base, NY, February, 1985

[8] McIntyre, J.R., and Adelman, L., *Senior Battle Staff Decision Aids: Final Technical Report*, Report No. PAR 85-111, Rome Air Development Center, Griffiss Air Force Base, NY, December, 1985

[9] *Senior Battle Staff Decision Aids*, Statement of Work, Rome Air Development Center, PR No. B-3-3603, contract no. F30602-83-C-0154, December 2, 1982

# SOFTWARE QUALITY MEASUREMENT - A METHODOLOGY AND ITS VALIDATION

FOURTH ANNUAL PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

November 10-11, 1986

James L. Warthman
COMPUTER SCIENCE INNOVATIONS, INC.

CSI

## NOTES:

## KEY FINDINGS

- SOFTWARE QUALITY MEASUREMENT (SQM) IS A GENERALLY SOUND TECHNIQUE FOR EXAMINING QUALITY ISSUES

- FRAMEWORK AND METHODOLOGY NEED IMPROVEMENTS

CSI

NOTES:

## AGENDA

- SQM BACKGROUND
  - SOFTWARE QUALITY MODEL
  - SQM METHODOLOGY
- SQM DEMONSTRATION STUDY
- FINDINGS & RECOMMENDATIONS
- SUMMARY

CSI

NOTES:

## SOFTWARE QUALITY MODEL

- OVERVIEW
  - IMPROVE SOFTWARE QUALITY
  - BEYOND SIMPLE QA CHECKLISTS
  - SPECIFY & MEASURE QUALITY
  - QUANTITATIVE APPROACH
  - EVOLVED OVER TEN YEARS

CSI

# NOTES:

## SOFTWARE QUALITY MODEL

### 13 SOFTWARE QUALITY FACTORS

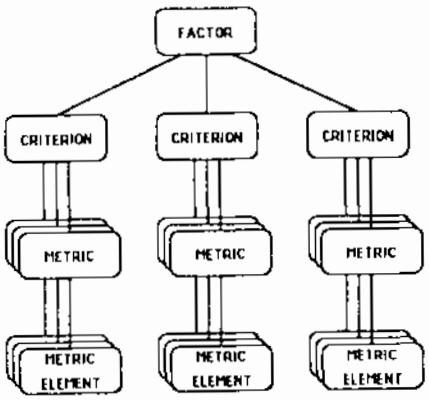
| PERFORMANCE | DESIGN | ADAPTATION |
|---|---|---|
| EFFICIENCY | CORRECTNESS | EXPANDABILITY |
| INTEGRITY | MAINTAINABILITY | FLEXIBILITY |
| RELIABILITY | VERIFIABILITY | INTEROPERABILITY |
| SURVIVABILITY | | PORTABILITY |
| USABILITY | | REUSABILITY |

CSI

# NOTES:

## SOFTWARE QUALITY MODEL

### HIERARCHICAL ATTRIBUTES

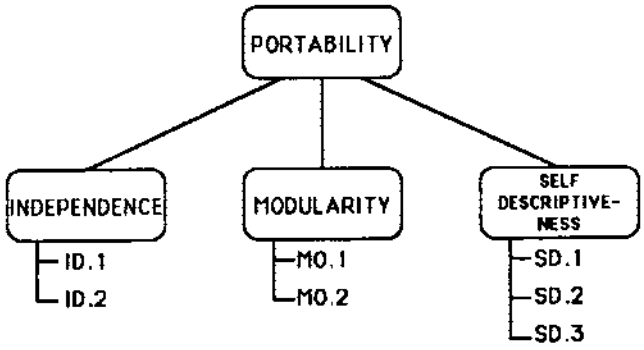



CSI

# NOTES:

## SOFTWARE QUALITY MODEL

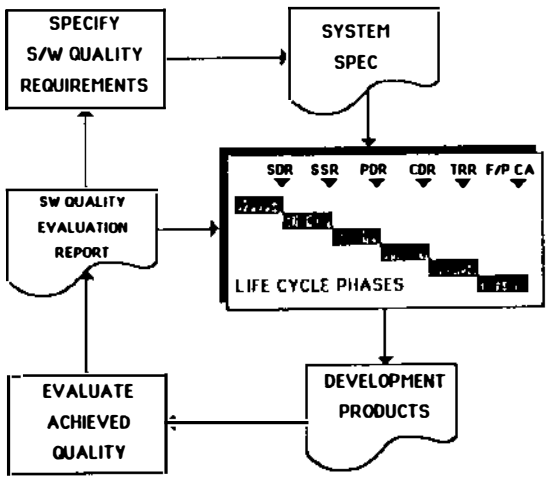
### DECOMPOSITION OF A QUALITY FACTOR




CSI

# NOTES:

## SQM METHODOLOGY




CSI

# NOTES:

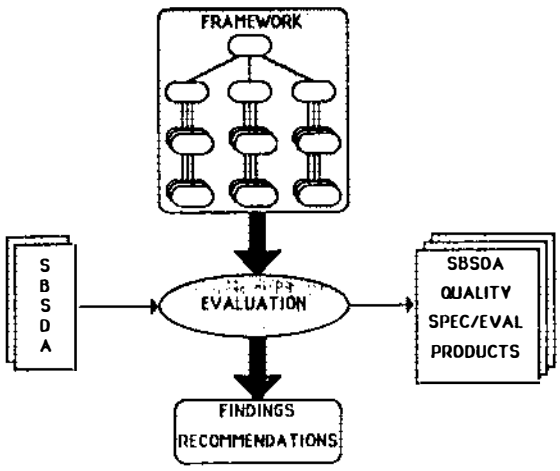
## AGENDA

- SQM BACKGROUND
- ➡ SQM DEMONSTRATION STUDY
  - OBJECTIVES
  - QUALITY SPECIFICATION APPROACH
  - QUALITY EVALUATION APPROACH
- FINDINGS & RECOMMENDATIONS
- SUMMARY

CSI

NOTES:

## OBJECTIVES




CSI

# NOTES:

## QUALITY SPECIFICATION APPROACH

### QUALITY SURVEY

| SOFTWARE QUALITY FACTOR / SYSTEM OR SOFTWARE-UNIQUE FUNCTION | PERFORMANCE | | | | | DESIGN | | | ADAPTATION | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | EFFICIENCY | INTEGRITY | RELIABILITY | SURVIVABILITY | USABILITY | CORRECTNESS | MAINTAINABILITY | VERIFIABILITY | EXPANDABILITY | FLEXABILITY | INTEROPERABILITY | PORTABILITY | REUSABILITY |
| DEVELOP KNOWLEDGE | A | - | A | - | A | A | G | A | E | E | A | G | E |
| MONITOR ENEMY PERFORMANCE | A | - | A | - | A | G | G | A | E | E | G | G | E |
| ANALYZE ENEMY PERFORMANCE | G | - | A | - | E | G | G | A | E | E | G | G | E |

**Factor Scores**

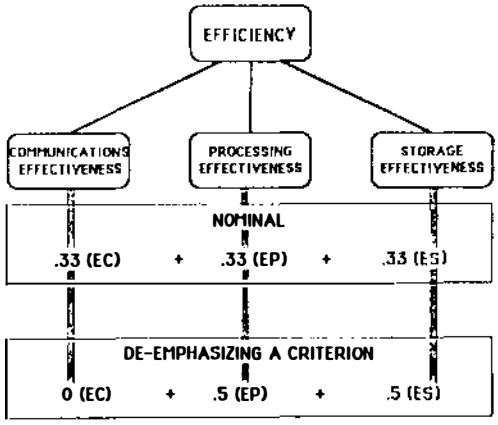
E = .90 - 1.0
G = .80 - .89
A = .70 - .79
- = not applicable

CSI

# NOTES:

## QUALITY SPECIFICATION APPROACH

### CRITERIA WEIGHTING

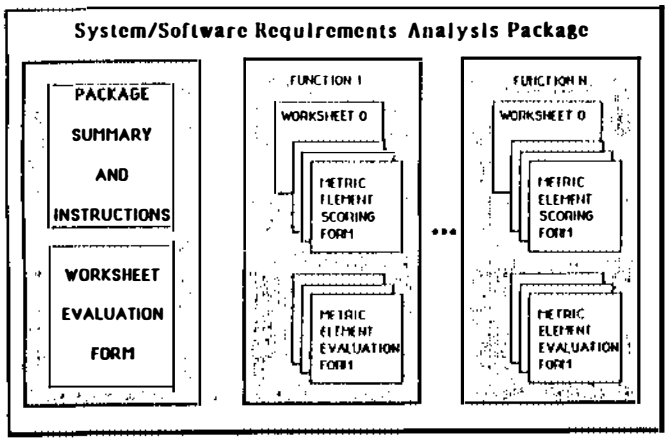



CSI

# NOTES:

## QUALITY EVALUATION APPROACH

### METRIC ELEMENT EVALUATION PACKAGE

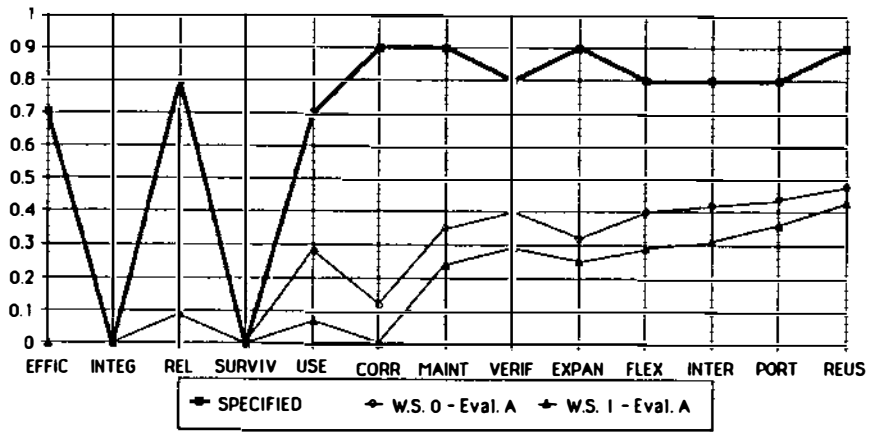



CSI

# NOTES:

## QUALITY EVALUATION APPROACH

### SPECIFIED .vs. MEASURED RESULTS




CSI

# NOTES:

## AGENDA

- SQM BACKGROUND
- SQM DEMONSTRATION STUDY
- ➡ FINDINGS & RECOMMENDATIONS
  - BRINGS FOCUS TO SOFTWARE QUALITY
  - IMPROVEMENTS ARE NEEDED
- SUMMARY

CSI

# NOTES:

## BRINGS FOCUS TO SOFTWARE QUALITY

- QUALITY AWARENESS IS INCREASED
  - USER'S NEEDS
  - LIFE CYCLE CONSIDERATIONS
- FEASIBILITY IS OBJECTIVELY EVALUATED
  - TECHNICAL
  - COST
- PROBLEMS CAN BE DETECTED EARLY
- RESULTS IN HIGHER QUALITY SOFTWARE
- LIFE CYCLE COSTS MAY BE REDUCED

CSI

# NOTES:

## IMPROVEMENTS ARE NEEDED

FINDING:

- QUALITY SPECIFICATION TOO SUBJECTIVE
  - *EXCELLENT* .vs. *GOOD* .vs. *AVERAGE*
  - FACTOR DEFINITIONS
  - POOR REPEATABILITY

RECOMMENDATIONS:

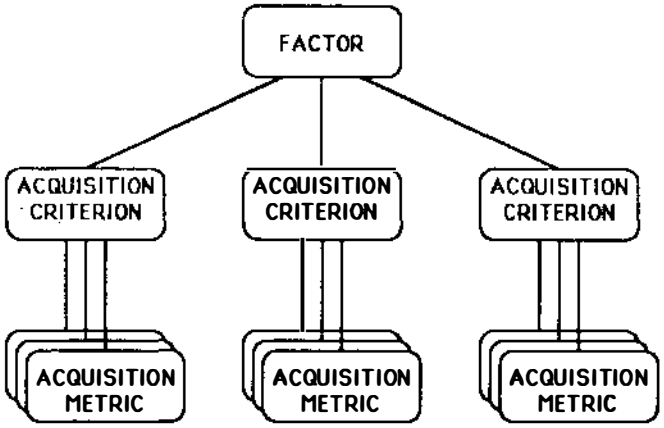
- NEW ACQUISITION HIERARCHY
  - CONCRETE ASPECTS OF QUALITY
  - USERS ANSWER SPECIFIC QUESTIONS
  - REDUCED LEARNING CURVE
  - INCREASED OBJECTIVITY

CSI

# NOTES:

## IMPROVEMENTS ARE NEEDED

### ACQUISITION HIERARCHY




CSI

# NOTES:

## IMPROVEMENTS ARE NEEDED

FINDING:

- INTERMEDIATE RESULTS, RATIONALE, TRADE-OFFS ARE NOT CAPTURED OR DOCUMENTED

RECOMMENDATIONS:

- ADD REPORT DEFINITION TO METHODOLOGY
- CAPTURE ALL CHANGES TO QUALITY REQUIREMENTS

CSI

NOTES:

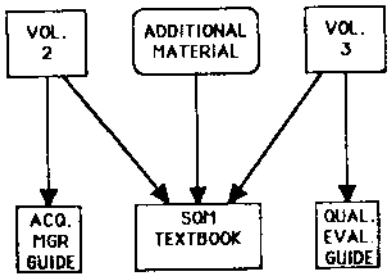
## IMPROVEMENTS ARE NEEDED

**FINDING:**

- **THEORY, EXAMPLES, PROCEDURES NOT WELL DIFFERENTIATED**

**RECOMMENDATIONS:**

- **RE-STRUCTURE GUIDEBOOKS**




CSI

# NOTES:

## IMPROVEMENTS ARE NEEDED

FINDING:

- FRAMEWORK HAS NOT ADAPTED TO RECENT TECHNOLOGICAL ADVANCES
  - FAULT TOLERANT SYSTEMS
  - 4TH GENERATION LANGUAGES
  - ADA

RECOMMENDATIONS:

- MODIFY DEFINITIONS AS NEEDED
  - FACTORS
  - CRITERIA
  - METRICS

CSI

NOTES:

## IMPROVEMENTS ARE NEEDED

**FINDING:**

- **HARD TO ANSWER WORKSHEET QUESTIONS**
  - **DIFFICULT TO LOCATE INFORMATION**
  - **DIFFERENTIATION BETWEEN *NO* AND *N/A* IS SUBJECTIVE**

**RECOMMENDATIONS:**

- **GROUP RELATED QUESTIONS**
- **ELIMINATE THE USE OF *N/A***

CSI

# NOTES:

## IMPROVEMENTS ARE NEEDED

FINDING:

- MISLEADING TO DRAW CONCLUSIONS FROM METRIC SCORES AND TRENDS
  - ACROSS WORKSHEETS
  - SPECIFIED .vs. MEASURED

RECOMMENDATIONS:

- EXPLAIN THE SUBJECTIVITY OF SCORES
- CONTINUE TO GATHER DATA
  - NORMALIZE SCORES
  - VALIDATE FRAMEWORK

CSI

# NOTES:

## AGENDA

- SQM BACKGROUND
- SQM DEMONSTRATION STUDY
- FINDINGS & RECOMMENDATIONS
- ➡ SUMMARY

CSI

NOTES:

## SUMMARY

- FRAMEWORK MUST EVOLVE
- GUIDEBOOKS SHOULD BE IMPROVED
- METHODOLOGY CAN BE USED NOW

CSI

# NOTES:

# A COMPARISON OF COUNTING METHODS FOR SOFTWARE SCIENCE AND CYCLOMATIC COMPLEXITY

Research and Development Engineer
Corvallis Workstation Operation
Nancy Currans

## ABSTRACT

It has been shown that about half of software product development time is spent in the testing phase, and that most of the dollars spent on a product are spent maintaining it.

This paper discusses a system of computing various software complexity metrics for large software projects to better pinpoint problem areas. It reviews McCabe's cyclomatic complexity and Halstead's laws. Problems that occur when calculating the metrics are discussed; the metrics are computed several different ways using data obtained from a large software project. The result of varying the counting methods is investigated.

A tool is described which enables complexity data to be easily compiled. An analysis is then done to determine which complexity metric best identifies problem areas in the code. A strong relationship is shown to exist between Halstead's laws, McCabe's cyclomatic complexity, lines of code, and the number of bugs reported in the project.

## BIOGRAPHICAL SKETCH

Nancy is a native Oregonian. She received the following degrees from Oregon State University: Business Administration (B.S., 1979); Computer Science (B.S., 1981; M.S., 1985). She worked as a Quality Engineer for Intel Corporation in Hillsboro, Oregon where her main responsibility was the design and implementation of a MTBF system for PCB products. She then worked at Hewlett Packard in Corvallis, Oregon as a Software Engineer, responsible for the design and implementation of a test system for a large software project. Nancy is a new mother and currently taking time off from her professional career to explore the institution of motherhood with her two children. She can be reached at:

Nancy Currans
3422 N.W. Maxine Circle
Corvallis, OR 97330
(503) 758-7151

## INTRODUCTION

This paper first discusses a tool that was used to extract complexity data from a 'C' program. Following, two of the more popular software complexity metrics are reviewed: McCabe's cyclomatic complexity, and Halstead's software science. The problems that occur when computing the metrics are discussed, and resolutions to those problems for the purpose of this study are given. Finally, the difficulty a programmer has working with a program is assessed by studying the relationship between the complexity data and the errors found in his code.

Although it is dangerous to draw conclusions from one specific study and then generalize them to all software, the results presented here support the conclusions of many published articles. McCabe's metrics and Halstead's laws are computed different ways; the results of the study show that the counting methods are strongly related to each other, and the statistical results are essentially the same no matter which counting method is used. The popular metric, lines of code without comments, has one of the strongest relationships to errors when compared to any of McCabe's or Halstead's metrics.

Further studies must be performed using code from other actual software projects for any conclusions to be drawn for all code developed and implemented in industry, but this study suggests that strong relationships do exist between bug rate and the metrics under study.

## THE REDUCED FORM METRIC DATA EXTRACTION TOOL

The correlation of error data to any complexity metric requires that there be an extraction technique to obtain the appropriate program characteristics. The output from the method used in this study is known as the Reduced Form (3). The program that creates the Reduced Form from 'C' source code will hence be referred to as RED. It was designed by two researchers who wanted to obtain data from industry without sacrificing company secrecy. Some of the characteristics of interest include information about program size, data structures, and the number and usage of local and global variables. The Reduced Form was designed to summarize information for each subroutine or function in the code system, but bar reproduction of the actual code.

The data extraction tool was implemented using LEX. It uses basic pattern matching techniques, and its output is composed of five parts:

1. An identification line which consists of the name of the subprogram.

2. A declaration table, which lists the number of times each type of declaration was used in the subprogram.

3. An operand table which lists an alias for each unique string, constant, and variable, and the number of times the token was used.

4. An operator table which lists the operators used and the function calls made, in addition to the number of times each occurred.

5. A length line, which indicates the number of source lines and the number of non-commentary source lines in the subprogram.

As suggested by Cook and Harrison (3), this tool does not provide the researcher with all the information about the code that he may be interested in. In fact, it is clear that the tool does not support every metric currently in use. It does not supply data flow or detailed topographical control flow information. It also does not address operators in multiple contexts. For example, the parentheses pair in a decision statement is treated the same as one used in an arithmetic calculation. Therefore, metrics that require nesting information, flow of data, or operator counts that are context sensitive cannot be derived from the Reduced Form.

## EXAMPLE OF INPUT TO AND OUTPUT FROM RED

Following is an example of some 'C' code that was input to RED: (3)

```
readfile (fname)
char *fname; {
        register FILE *f = fopen (fname,"r");
        if (f==0) {
                error ("Can't read %s",fname);
                return;
        }
        erasedb ();
        while (fgets(line,sizeof(line),f)) {
                linelim = 0;
                if (line[0] != '#') yyparse ();
                }
        fclose (f);
        DBchanged = 0;
        linelim = -1;
        }
```

Following is the output from RED when the preceding code was input:

```
PROCEDURE readfile()
DCLS
FILE     1
char     1
register 1
CONSTANTS
CON000020 4
CON000021 1
VARIABLES
VAR 000131 1 unknown unknown
VAR000128  4 FILE  local
VAR000127  4 FILE  formal parameter
VAR000129  3 unknown unknown
VAR000008  2 int global
STRINGS
STR000047  1
STR000046  1
STR000048  1
FNCALLS
erasedb()  1
error()    1
fclose()   1
fgets()    1
fopen()    1
yyparse()  1
OPERATORS
!= 1
== 2
" 1
* 2
' 4
- 1
; 10
= 4
== 1
[ 1
if() 2
return 1
sizeof 1
while() 1
{ 3
LENGTH 16 16
```

## McCABE AND CYCLOMATIC COMPLEXITY

The metric McCabe developed (4), the cyclomatic complexity v(G), is based on the program control graph. A program control graph is a directed graph that represents the flow of control through a piece of code. It has unique entry and exit nodes. Each node corresponds to a sequential block of code that can only be entered at

the first statement, exited at the last statement, and has no internal transfer of control. Each arc represents the flow of control between blocks. It is assumed that each node can be reached from the entry node, and that the exit node can be reached from any other node.

A formal definition of v(G) follows: v(G) = e - n + (2 x p), where:

- e: the number of edges in the graph
- n: the number of nodes in the graph
- p: the number of connected components in the graph

McCabe showed that the cyclomatic complexity of a structured program equals the number of predicates in the code plus one. The cyclomatic complexity of a CASE statement adds one less than the number of cases to the computation. This is because any N-way CASE statement can be simulated by N-1 IF statements. N-way computed GO TO statements also add N-1 to v(G). Hence, the calculation of v(G) can be made entirely from the syntactic characteristics of the program. This is a much simpler method of computing v(G) than counting nodes, edges, and connected components in the program control graph.

Myers (5) extended McCabe's work; he considered compound predicates. The problems with computing cyclomatic complexity, such as how to handle a program with several subprograms, or counting predicates versus simple predicates, as well as the relationship between v(G) and the number of problems reported will be discussed in the section on "ANALYSIS OF DATA".

## HALSTEAD AND SOFTWARE SCIENCE

Maurice Halstead (2) hypothesized that there are a set of invariant laws that may be applied to algorithms. These can be compared to the physical laws of science. His theories have become known as "software science". The laws of software science are said to hold true for all implementations of algorithms.

Halstead's premise was that regardless of the language used, implementations of algorithms have physical characteristics that can be measured and calculated from four basic program measurements:

- n1: the number of distinct operators in a program
- n2: the number of distinct operands in a program
- N1: the total number of occurrences of operators in a program
- N2: the total number of occurrences of operands in a program

The source code can be parsed into "tokens", which are divided between operators and operands. Operands include constants and variables. Operators are those tokens in the program that affect the ordering or value of the operands.

The laws that Halstead proposed based on n1, n2, N1, and N2, are as follows:

- VOCABULARY (n) is defined as the total number of distinct tokens in a

program: n = n1 + n2.

• LENGTH (N) is defined as the total number of tokens in a program: N = N1 + N2.

• LENGTH ESTIMATOR (Nhat) is defined to be strongly dependent upon n1 and n2: Nhat = [n1 x log2 (n1)] + [n2 x log2 (n2)].

• VOLUME (V) is defined as the number of bits needed to represent the program: V = N x log2 (n).

• POTENTIAL VOLUME (V*) is defined as the most efficient implementation of an algorithm in a given language.

• PROGRAM LEVEL (L) relates the volume (V) to the potential volume (V*) of a program implementation in the following way: L = V* / V.

• PROGRAM LEVEL ESTIMATOR (Lhat) is defined as: Lhat = (2 / n1) x (n2 / N2).

• DIFFICULTY (D) is defined to be inversely proportional to the program level because as the program volume increases, the "difficulty" of the program implementation increases: D = 1 / L or D = V / V*.

• EFFORT (E) is defined as the amount of work needed to implement an algorithm: E = V / L or E = [N x log2 (n)] / L.

The laws that Halstead proposed were intended to be tested, somewhat like the physical laws of science. He suggested many relationships between the metrics themselves, and between the metrics and actual implementations of algorithms.

This paper computes Halstead's metrics several different ways. The problems with Halstead's laws and their computation, such as how pairs of operators (2-word key words, parentheses, brackets), and global versus local variables should be counted, along with the relationship between the metrics and the number of bugs reported will be discussed in the the section on "ANALYSIS OF DATA".

## THE FIELD STUDY

The code obtained for this study was written in 'C' to run on a unix operating system. The modules of code were broken down according to user functionality; some examples included I/O, math functions, and graphics commands.

The software complexity metrics applied in this study assume that modules receive the same amount of initial testing or debugging prior to entering the testing phase, and that all modules receive the same amount of testing while error data is being collected. This is consistent with the requirements suggested by Ottenstein when performing this type of study (6). Modules not meeting both criteria were not considered.

### Collecting the Error Data

Notesfiles were used to collect the data as discussed by Staley (9). Bugs were posted to the appropriate notesfile as individual basenotes. Each basenote contained the following information: a unique number identifying the bug report, the name of the engineer reporting the problem, the date the problem was reported, a description of the system being used when the problem occurred, the configuration of that system, the version of the software being used, and a detailed description of the problem.

Bugs that were fixed had a response attached to the basenote. The information in the response that was utilized in this study is as follows: the name of the engineer fixing the problem (or the name of the engineer posting the response), the names of the modules affected by the change, and the severity of the problem, categorized as critical, severe, moderate, or trivial.

### Collecting the Complexity Data

RED was used (see "THE REDUCED FORM DATA EXTRACTION TOOL") to tokenize the 'C' source code. Another program was written to compute the metrics for the purpose of this study.

## ANALYSIS OF DATA

This study repeated software complexity metric studies found in recent publications. The study was unique in that the 'C' programming language was used. No previous publications studied the relationship between Halstead's laws or McCabe's metrics and 'C', and most of the studies used either FORTRAN or COBOL programs from a text book or controlled experimentation, rather than code from industry.

Before discussing any results, a short description of bug rates is needed. The bug rates in each module can be calculated in several different ways. Recall that there are four categories of bugs: critical, serious, moderate, and trivial. It is possible to perform statistical analysis using the total number of bugs found in each module, or to group them by category. Rather than choose one particular method of grouping the bugs, several different groupings were used. The total number of bugs, the total number of non-trivial bugs, the number of critical and serious bugs taken together, and then each category by itself is considered.

In the tables below, the following abbreviations for the severity of the bugs are used: "C" for critical, "S" for severe, "M" for moderate, and "T" for trivial. Table 1 shows the total number of errors reported against each of the thirty modules of code by severity:

| module | C | S | M | T | total |
|---|---|---|---|---|---|
| mod1 | 1 | 3 | 1 | 2 | 7 |
| mod2 | 3 | 6 | 20 | 6 | 35 |
| mod3 | 5 | 9 | 8 | 5 | 27 |
| mod4 | 0 | 1 | 0 | 0 | 1 |
| mod5 | 1 | 2 | 1 | 6 | 10 |
| mod6 | 0 | 1 | 3 | 2 | 6 |
| mod7 | 0 | 0 | 0 | 0 | 0 |
| mod8 | 0 | 0 | 1 | 0 | 1 |
| mod9 | 0 | 1 | 1 | 0 | 2 |
| mod10 | 3 | 8 | 5 | 5 | 21 |
| mod11 | 1 | 3 | 6 | 3 | 13 |
| mod12 | 1 | 4 | 11 | 5 | 21 |
| mod13 | 3 | 3 | 6 | 1 | 13 |
| mod14 | 0 | 1 | 0 | 0 | 1 |
| mod15 | 0 | 3 | 0 | 0 | 3 |
| mod16 | 1 | 0 | 2 | 0 | 3 |
| mod17 | 3 | 5 | 6 | 1 | 15 |
| mod18 | 4 | 8 | 11 | 3 | 26 |
| mod19 | 0 | 0 | 0 | 0 | 0 |
| mod20 | 0 | 0 | 0 | 0 | 0 |
| mod21 | 0 | 0 | 0 | 0 | 0 |
| mod22 | 7 | 17 | 18 | 2 | 44 |
| mod23 | 0 | 3 | 2 | 0 | 5 |
| mod24 | 0 | 0 | 0 | 0 | 0 |
| mod25 | 3 | 2 | 2 | 2 | 9 |
| mod26 | 1 | 2 | 0 | 0 | 3 |
| mod27 | 0 | 1 | 8 | 5 | 14 |
| mod28 | 0 | 0 | 6 | 10 | 16 |
| mod29 | 1 | 2 | 0 | 0 | 3 |
| mod30 | 0 | 1 | 3 | 0 | 4 |

Table 1

**McCabe's Cyclomatic Complexity Versus Bug Rate**

It is not clear how to compute v(G) for a module with several routines. Should, as McCabe suggested, v(G) be the sum of the v(G)s for each routine within a module, or should routine boundaries be ignored? Partial v(G)s can be computed for each routine, and then summed across the module to compute the module's cyclomatic complexity. The alternative is to compute the module's v(G) without regard to routine boundaries.

Myers (5) extended McCabe's original work for complex conditions. He felt that v(G) should be equal to the number of predicates and conditionals, plus one. Consider a module with entirely straight-line code except for the following line: "IF a<b AND b>c THEN...". If v(G) is defined as the number of decisions plus one, then the cyclomatic complexity equals 2. If, on the other hand, it is defined to include the number of conditions, v(G) equals 3.

Four different schemes for counting complex conditionals were studied and computed for the program modules:

1. v(G) was computed as the number of predicates in the module plus one. Each of the following keywords occurring in the module added one to the complexity count: WHILE, IF-THEN, IF-THEN-ELSE, and FOR. N-way CASE statements increased the count by N-1.

2. v(G) was computed as one added to the number of simple predicates and conditions, where the predicates were as specified in (1) above. (CASE statements were also handled as described above.)

3. A "partial" v(G) was computed for each routine in the module, as described in (1) above. The cyclomatic complexity was derived by the summation of all "partial" v(G) values.

4. "Partial" v(G)s were computed for each routine as described in (2) above. A summation of the "partial" v(G)s was taken to obtain the final v(G) value.

The first two methods give no consideration to routine boundaries. The last two calculate v(G) for each routine, and use the summation to compute the cyclomatic complexity.

A comparison was done between the four counting methods with the various bug groupings. The correlations are given in Table 2:

| metric | CSMT | CSM | CS | C | S | M | T |
|---|---|---|---|---|---|---|---|
| v1(G) | .763 | .771 | .839 | .755 | .854 | .599 | .330 |
| v2(G) | .776 | .784 | .840 | .766 | .851 | .623 | .340 |
| v3(G) | .745 | .752 | .828 | .751 | .840 | .574 | .329 |
| v4(G) | .767 | .773 | .835 | .763 | .844 | .607 | .346 |

Table 2

The correlation coefficients suggest a strong relationship between the bug rate and v(G). The relationship between v(G) and the moderate or trivial bugs is not strong. The relationship between the bug rate and the critical and serious bugs taken together, or separately is very strong.

An observation can be made that all of the methods of computing cyclomatic complexity show a strong relationship to bug rate. Therefore, correlation coefficients were computed between the four metrics to establish whether there is a relationship between the counting methods:

| metric | v2(G) | v3(G) | v4(G) |
|---|---|---|---|
| v1(G) | .995 | .996 | .993 |
| v2(G) | .--- | .993 | .998 |
| v3(G) | .--- | .--- | .996 |

Table 3

The high correlation coefficients suggest that the four variations of computing the cyclomatic complexity are very strongly related. Table 2 shows that the correlations for all of the counting methods with the bug rate are very close to the

same. Table 3 shows that a strong relationship exists between all methods of computation. Together, the tables indicate that there is no difference between the four methods of computation, and therefore suggest that, if v(G) is to be computed, the easiest method should be used.

**Halstead's Software Science Versus Bug Rate**

The most basic problem with Halstead's software science computations (7) is the ambiguity of the counting rules when deriving n1, n2, N1, and N2. For instance, whether or not declarations should impact calculations. He relates his method of counting to implementations of algorithms. He contended that declarations have nothing to do with the algorithm itself, and hence should not impact the calculations. Since he worked with FORTRAN, which has implicit declarations, the question was not a serious one in his studies.

Another question concerns counting local versus global variables. If a variable is declared global and then used locally within a routine, or set of routines, is it counted as the same variable, or as a distinctly different variable?

It is not clear if counting should be done by functional module or by individual routine. Should n1, n2, N1, and N2 be calculated for each routine in each module? Potentially, each routine uses the same operators; should they be counted as unique for each routine? How does one combine measures to obtain one set of values for each module?

Even more elementary problems exist in defining an unambiguous counting strategy. How are operators that occur in pairs, such as IF-THEN, {}, and () counted? Is each half counted as a distinct operator, or are they grouped together as one? Should delimiters be counted, such as the semi-colon?

To solve the ambiguity problem for the purpose of this study, several counting methods were defined. The list of rules followed for all counting strategies were:

1. Count only executable statements. This counting rule was used primarily because Halstead based his theories on implementations of algorithms. No consideration was given to the language-dependent overhead of implementing the algorithm in a given language. (i.e. declaration of variables.)

2. Any pairs of symbols, such as parentheses, are counted together as one. They function as a single operator.

3. Count the semi-colon as a unique operator.

4. Count the tokens (operators and operands) the same in any context. For example, the parentheses pair in a decision statement is no different from one used in an arithmetic calculation.

5. In the GO TO <label> statement, count the GO TO as an operator, and the <label> as an operand. This is different from Halstead's original treatment of each GO TO <label>. He counted each occurrence of GO TO <label> as a unique operator.

6. Count function calls as operators.

Several variations of counting methods were used in this study:

1. n1 and n2 were calculated by computing the n1 and n2 values for each routine, and then summing them over the entire module. In other words, routine boundaries were considered. An operator or operand occurring in two different routines whether local or global in scope, was counted as unique in each of the routines. n1 included the distinct number of functions called, keywords, and arithmetic or logical operators. n2 included all global and local variables, labels, and constants. N1 and N2 were calculated by counting the total number of occurrences of operators and operands, respectively, over the entire module.

2. Method 2 was similar to method 1 except that it did not consider routine boundaries. If a variable was global, it was counted only once in n2. (In method 1, a global variable occurring in routines "a", "b", and "c" would add one to the n2 value in each of the three routines. Ultimately, 3 would be added to n2.) Operators were handled similarly. A minus sign added one to n1, regardless of how many routines it occurred in. Notice that N1 and N2 are identical for methods 1 and 2. They represent the total number of occurrences of tokens, regardless of how "distinctness" is defined.

3. Method 3 was an attempt to remove one variable from the calculations. Prior to any coding, if the number of operators used in the program can be estimated, the only unknown needed to calculate Nhat is the number of distinct operands that will be used in the code. After the design phase of the project, an estimation of the number of local and global variables, labels, and constants needed can be made. Method 3 tests this theory by calculating n2, as per the description in method 2; it uses the constant 40 for n1. Forty was an estimate obtained by counting and averaging the number of distinct operators used in random samples of 'C' code.

Halstead (1) studied the relationship between his length estimator (Nhat) and his length metric (N). Using polished code, he asserted that Nhat is a good estimator of N, providing the code is "pure". Following are the results of correlating N with Nhat:

| metric | N1 | N2 | N3 |
|---|---|---|---|
| Nhat | .976 | .971 | .951 |

Table 4

The results from this study support the assertion that there is a high correlation between Halstead's length and his length estimator, no matter which counting method is used.

To test for a relationship between the length or length estimator and the bug rate, the following correlation coefficients were computed:

| metric | CSMT | CSM | CS | C | S | M | T |
|---|---|---|---|---|---|---|---|
| N1 | .761 | .750 | .796 | .749 | .794 | .604 | .407 |
| N2 | .761 | .750 | .796 | .749 | .794 | .604 | .407 |
| N3 | .761 | .750 | .796 | .749 | .794 | .604 | .407 |
| Nhat1 | .717 | .703 | .779 | .719 | .784 | .532 | .398 |
| Nhat2 | .702 | .677 | .736 | .692 | .734 | .527 | .435 |
| Nhat3 | .698 | .675 | .743 | .685 | .749 | .516 | .424 |

Table 5

A strong relationship exists between both the length and the bug rates, and the length estimator and the bug rates. There is a strong relationship between the metrics and the more significant bugs; the metrics are not strongly related to the trivial or moderate bugs taken by themselves. The relationship between all variations of computing length N and the bug rates are identical. This is to be expected from the counting rules. The number of tokens in the code for each of the three counting methods (N1, N2, and N3) is the same.

Correlation coefficients were calculated between Halstead's volume and the bug rate. Table 6 shows the correlation coefficients:

| metric | total bugs | CSM | CS | C | S | M | T |
|---|---|---|---|---|---|---|---|
| V1 | .763 | .753 | .803 | .756 | .802 | .601 | .406 |
| V2 | .763 | .750 | .799 | .754 | .796 | .601 | .410 |
| V3 | .763 | .752 | .802 | .754 | .800 | .601 | .601 |

Table 6

Although no specific studies were found relating volume to the bug rate, it is interesting to note that these include among the highest correlations found in the study. The critical and serious bugs taken together have the strongest relationship to volume. The moderate and trivial bug count is not related to the volume metric.

Shen, Conte, and Dunsmore (8) studied the relationship between difficulty and the bug rate. Their studies ranked modules according to their error-proneness, and observed whether the difficulty measure increased accordingly. Correlation coefficients in Table 7 were computed to study the relationship between the three difficulty measures and the bug rate.

| metric | CSMT | CSM | CS | C | S | M | T |
|---|---|---|---|---|---|---|---|
| D1 | .724 | .712 | .769 | .740 | .759 | .559 | .397 |
| D2 | .575 | .563 | .556 | .591 | .522 | .495 | .321 |
| D3 | .268 | .280 | .238 | .249 | .225 | .287 | .042 |

Table 7

The counting method matters when computing the difficulty. The first method shows the strongest relationship to the bug rate. The other two methods do not indicate any relationship at all.

Of all Halstead's metrics, effort (E) is most often correlated to bug rate. The correlation coefficients suggest a strong relationship between effort and bug rate in Table 8.

| metric | CSMT | CSM | CS | C | S | M | T |
|---|---|---|---|---|---|---|---|
| E1 | .722 | .712 | .778 | .742 | .772 | .551 | .389 |
| E2 | .696 | .689 | .691 | .686 | .673 | .596 | .367 |
| E3 | .760 | .761 | .784 | .753 | .775 | .636 | .364 |

Table 8

The strongest relationships exist between effort and the more significant bugs (critical and serious). Strong relationships do not exist between the moderate or trivial bugs and the effort.

**Comparative Study Between Complexity Metrics**

Lines of code is sometimes used as a simple complexity metric. Therefore, this study investigated its relationship to bug rate. Below is a table of correlation coefficients that show this relationship. Lines of code has been computed two ways. "LOCw" is the number of lines of code including comments, and "LOCwo" is the number of lines of code excluding comments. For the purpose of this study, if a line of code has comments on it, that line is counted in both the measures. The only line of code that is not considered in "LOCwo" is source code that only has a comment on it.

| metric | CSMT | CSM | CS | C | S | M | T |
|---|---|---|---|---|---|---|---|
| LOCw | .787 | .781 | .770 | .716 | .773 | .687 | .424 |
| LOCwo | .826 | .819 | .830 | .772 | .833 | .699 | .439 |

Table 9

Lines of code, counted with and without the comments, correlated well to the number of bugs in the module. Lines of code without comments has among the strongest relationship to bug rate when compared with any of Halstead's or McCabe's metrics.

Going one step further, the relationship between all of the metrics, computed using method one was investigated. Table 10 shows the correlation coefficients:

| metric | LOCwo | v(G) | N | Nhat | V | D | E |
|---|---|---|---|---|---|---|---|
| LOCw | .982 | .896 | .907 | .883 | .909 | .890 | .975 |
| LOCwo | .--- | .818 | .903 | .799 | .832 | .820 | .810 |
| v(G) | .--- | .--- | .973 | .960 | .972 | .950 | .911 |
| N | .--- | .--- | .--- | .976 | .998 | .989 | .948 |
| Nhat | .--- | .--- | .--- | .--- | .983 | .974 | .959 |
| V | .--- | .--- | .--- | .--- | .--- | .990 | .962 |
| D | .--- | .--- | .--- | .--- | .--- | .--- | .964 |

Table 10

There is a strong relationship between all three metrics: lines of code, Halstead's laws, and McCabe's cyclomatic complexity.

## CONCLUSIONS

RED, the Reduced Form tool, proved to be invaluable in this study. It made data extraction simple; the only work involved once the metrics were defined, was writing the program to analyze the data. With the use of this tool, many metrics and derivations of those metrics could easily be computed. Although it does not provide all the information needed to compute any metric, it does provide researchers a way of obtaining a lot of information easily.

All of the metrics studied in this report correlate well to the number of bugs reported against the code: lines of code and variations on both McCabe's cyclomatic complexity and Halstead's laws. The relationship between the metrics and the bugs reported is strongest for the severe and critical bugs.

Comparing the correlations between the variations of the metrics with the bug rates, very little difference was found. No one counting method stood out as being more strongly related to the bug rate than any other.

This paper shows that for the metrics studied, lines of code and cyclomatic complexity have the strongest relationship to bug rate. Since lines of code is so easy to compute, this study suggests that it should be used over the other metrics as a bug rate indicator.

## FOOTNOTES

1. Fitzsimmonds, Ann and Tom Love, "A Review and Evaluation of Software Science," COMPUTING SURVEYS, Vol. 10, No.1, March 1978.

2. Halstead, Maurice H. ELEMENTS OF SOFTWARE SCIENCE, Elsevier North-Holland. Inc., New York, 1977.

3. Harrison, Warren, and Curtis Cook. "A Method of Sharing Industrial Software Complexity Data," SIGPLAN Notices, Volume 20 Number 2, February 1985.

4. McCabe, Thomas, "A Complexity Measure", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-2, No. 4. December 1976.

5. Myers, Glenford J. "An Extension to the Cyclomatic Measure of Program Complexity", SIGPLAN Notices, October 1977.

6. Ottenstein, Linda M. "Quantitative Estimates of Debugging Requirements", IEEE Transactions on Software Engineering, Vol. SE-5, No. 5, September 1979.

7. Salt, Norman F. "Defining Software Science Counting Strategies", Department of Measurement, Evaluation and computer Applications, The Ontario Institute for Studies in Education, SIGPLAN Notices, March 1982.

8. Shen, Vincent Y, Samuel D Conte, and H. E. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support," Transactions on Software Engineering, Vol. SE-9, No. 2, March 1983.

9. Staley, Gordon. "A Unix-Based Software Development Problem Tracking System", SOFTWARE PRODUCTIVITY CONFERENCE PROCEEDINGS, pages 2-68 through 2-75, April 9-11, 1985.

**REFERENCES**

Bailey, C. T. and W. L. Dingee, "A Software Study Using Halstead Metrics," ACM Sigmetrics, pages 189-197, 1981.

Boehm,B.W. "Software and it's Impact: A Quantitative Assessment," DATAMATION, Vol. 19, pp. 48-49, May 1983.

Bulut, N., M. Halstead, and R. Bayer, "The Experimental Verification of a Structural Property of FORTRAN Programs," in Proceedings of the ACM Annual Conference, 1974, New York.

Fitzsimmonds, Ann and Tom Love, "A Review and Evaluation of Software Science," COMPUTING SURVEYS, Vol. 10, No.1, March 1978.

Funami Y. and Halstead, M. H. "A Software Physics Analysis of Akiyama's Debugging Data", CSD-TR144, Purdue University, Lafayette, Indiana, May 1975.

Gordon, R. D. and M. H. Halstead, "An Experiment Comparing FORTRAN Programming Times with the Software Physics Hypothesis," in 1976 Fall Joint Computer Conference, AFIPS Conference Proceedings, Vol. 45, Montvale, New Jersey: AFIPS Press, 1976, pages 936-937.

Halstead, Maurice H. ELEMENTS OF SOFTWARE SCIENCE, Elsevier North-Holland. Inc., New York, 1977.

Halstead, Maurice H., "Natural Laws Controlling Algorithm Structure?," SIGPLAN Notices, February 1972.

Harrison, Warren, and Curtis Cook. "A Method of Sharing Industrial Software Complexity Data," SIGPLAN Notices, Volume 20 Number 2, February 1985.

McCabe, Thomas, "A Complexity Measure", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-2, No. 4. December 1976.

Myers, Glenford J. "An Extension to the Cyclomatic Measure of Program Complexity", SIGPLAN Notices, October 1977.

Ottenstein, Linda M. "Quantitative Estimates of Debugging Requirements", IEEE Transactions on Software Engineering, Vol. SE-5, No. 5, September 1979.

Salt, Norman F. "Defining Software Science Counting Strategies", Department of Measurement, Evaluation and computer Applications, The Ontario Institute for Studies in Education, SIGPLAN Notices, March 1982.

Shen, Vincent Y, Samuel D Conte, and H. E. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support," Transactions on Software Engineering, Vol. SE-9, No. 2, March 1983.

Staley, Gordon. "A Unix-Based Software Development Problem Tracking System", SOFTWARE PRODUCTIVITY CONFERENCE PROCEEDINGS, pages 2-68 through 2-75, April 9-11, 1985.

# A COMPARISON OF COUNTING METHODS FOR CYCLOMATIC COMPLEXITY AND SOFTWARE SCIENCE

- Reduced Form Metric Data Extraction Tool (RED)
- McCabe's Cyclomatic Complexity
- Halstead's Software Science
- Analysis of Data
- Summary and Conclusions


# RED

- Developed at Oregon State University
- Extracts Data from "C" Source Code
- Produces the "Reduced Form"
- Composed of Five Parts

# REDUCED FORM

- **Identification Line:** Subprogram name
- **Declaration Table:** Number of occurrences each type is used
- **Operand Table:** Aliased constants, variables and strings
- **Operator Table:** Operators used and function calls made
- **Length Line:** Number of source lines with and without comments



# EXAMPLE OF INPUT TO RED

```
readfile (fname)
char *fname;{
        register FILE *f = fopen (fname,"r");
        if (f==0) {
                error ("Can't read %s",fname);
                return;
        }
        erasedb ();
        while (fgets(line,sizeof (line),f)){
                linelim = 0;
                if (line[0]!='#') yyparse ();
                }
        fclose (f);
        DBchanged = 0;
        linelim = -1;
        }
```

# EXAMPLE OF OUTPUT FROM RED

```
PROCEDURE readfile()                    FNCALLS
DCLS                                    erasedb() 1
FILE 1                                  error() 1
char 1                                  fclose() 1
register 1                              fgets() 1
CONSTANTS                               fopen() 1
CON000020 4                             yyparse() 1
CON000021 1                             OPERATORS
VARIABLES                               !=1
VAR000131 1 unknown unknown             ==2
VAR000128 4 FILE local                  "1
VAR000127 4 FILE formal parameter       *2
VAR000129 3 unknown unknown             .4
VAR000008 2 int global                  -1
STRINGS                                 ;10
STR000047 1                             =4
STR000046 1                             ==1
STR000048 1                             [1
                                        if() 2
                                        return 1
                                        sizeof 1
                                        while() 1
                                        { 3
                                        LENGTH 16 16
```

PORTABLE COMPUTERS

# MCCABE'S CYCLOMATIC COMPLEXITY

- Based on Program Control Graph
- v(G) = e − n +2p
- Number of Decisions + 1
- Exceptions: CASE and COMPUTED GOTO
- Suggested Upperbound is 10

PORTABLE COMPUTERS

## MCCABE'S CYCLOMATIC COMPLEXITY: PROBLEMS AND RESOLUTIONS

* Conceptual Problems
* Counting Problems
  . Routine vs. Modular Computation
  . Predicates vs. Simple Predicates
* Problem Resolutions
  . Four Counting Schemes
  . Permutations of Problems



## HALSTEAD'S SOFTWARE SCIENCE

* Based on Four Program Measurements:
  . n1: number of distinct operators
  . n2: number of distinct operands
  . N1: total number of operators
  . N2: total number of operands
* Computations from Program Characteristics
  . Vocabulary
  . Length
  . Length Estimator
  . Volume
  . Difficulty
  . Effort


## HALSTEAD'S SOFTWARE SCIENCE: PROBLEMS AND RESOLUTIONS

- Conceptual Problems
- Questionability of His Studies
- Counting Problems
  - . Declarations
  - . Local vs. Global Variables
  - . Routine vs. Modular Computation
  - . Accounting for Pairs of Operators
- Problem Resolutions
  - . Establishment of Basic Counting Rules
  - . Definition of Three Counting Methods



## MCCABE'S CYCLOMATIC COMPLEXITY VERSUS BUG RATE

| metric | CSMT | CSM | CS | C | S | M | T |
|---|---|---|---|---|---|---|---|
| v1(G) | .763 | .771 | .839 | .755 | .854 | .599 | .330 |
| v2(G) | .776 | .784 | .840 | .766 | .851 | .623 | .340 |
| v3(G) | .745 | .752 | .828 | .751 | .840 | .574 | .329 |
| v4(G) | .767 | .773 | .835 | .763 | .844 | .607 | .346 |

| metric | v2(G) | v3(G) | v4(G) |
|---|---|---|---|
| v1(G) | .995 | .996 | .993 |
| v2(G) | .--- | .993 | .998 |
| v3(G) | .--- | .--- | .996 |

# HALSTEAD'S SOFTWARE SCIENCE VERSUS BUG RATE

| metric | N1 | N2 | N3 |
|---|---|---|---|
| Nhat | .976 | .971 | .951 |

| metric | CSMT | CSM | CS | C | S | M | T |
|---|---|---|---|---|---|---|---|
| N1 | .761 | .750 | .796 | .749 | .794 | .604 | .407 |
| N2 | .761 | .750 | .796 | .749 | .794 | .604 | .407 |
| N3 | .761 | .750 | .796 | .749 | .794 | .604 | .407 |
| Nhat1 | .717 | .703 | .779 | .719 | .784 | .532 | .398 |
| Nhat2 | .702 | .677 | .736 | .692 | .734 | .527 | .435 |
| Nhat3 | .698 | .675 | .743 | .685 | .749 | .516 | .424 |

| metric | Total Bugs | CSM | CS | C | S | M | T |
|---|---|---|---|---|---|---|---|
| V1 | .763 | .753 | .803 | .756 | .802 | .601 | .406 |
| V2 | .763 | .750 | .799 | .754 | .796 | .601 | .410 |
| V3 | .763 | .752 | .802 | .754 | .800 | .601 | .601 |



# HALSTEAD'S SOFTWARE SCIENCE AND LINES OF CODE VERSUS BUG RATE

| metric | CSMT | CSM | CS | C | S | M | T |
|---|---|---|---|---|---|---|---|
| D1 | .724 | .712 | .769 | .740 | .759 | .559 | .397 |
| D2 | .575 | .563 | .556 | .591 | .522 | .495 | .321 |
| D3 | .268 | .280 | .238 | .249 | .225 | .287 | .042 |

| metric | CSMT | CSM | CS | C | S | M | T |
|---|---|---|---|---|---|---|---|
| E1 | .722 | .712 | .778 | .742 | .772 | .551 | .389 |
| E2 | .696 | .689 | .691 | .686 | .673 | .596 | .367 |
| E3 | .760 | .761 | .784 | .753 | .775 | .636 | .364 |

| metric | CSMT | CSM | CS | C | S | M | T |
|---|---|---|---|---|---|---|---|
| LOCw | .787 | .781 | .770 | .716 | .773 | .687 | .424 |
| LOCwo | .826 | .819 | .830 | .772 | .833 | .699 | .439 |

## COMPARATIVE STUDY BETWEEN COMPLEXITY METRICS

| metric | LOCwo | v(G) | N | Nhat | V | D | E |
|---|---|---|---|---|---|---|---|
| LOCw | .982 | .896 | .907 | .883 | .909 | .890 | .975 |
| LOCwo | – | .818 | .903 | .799 | .832 | .820 | .810 |
| v(G) | – | – | .973 | .960 | .972 | .950 | .911 |
| N | – | – | – | .976 | .998 | .989 | .948 |
| Nhat | – | – | – | – | .983 | .974 | .959 |
| V | – | – | – | – | – | .990 | .962 |
| D | – | – | – | – | – | – | .964 |


## SUMMARY AND CONCLUSIONS

* RED Proved to be Invaluable
* All Metrics Studied Correlate Well to Reported Bugs
* Metrics Correlate Best to the More Significant Bugs
* All Metrics Strongly Related to Each Other
* No "Best" Counting Method; Lines of Code without Comments Suggested

# Session 5

## RELEASE/VERSION CONTROL

*"Data Control in a Maintenance Environment"*
Michael W. Evans, Expertware, Inc.

*"Proposal for a Software Design Control System (DCS)"*
Robert Babb II, Richard Hamlet, Oregon Graduate Center

*"A Programmer's Database for Reusable Modules"*
T. G. Lewis, Oregon State University, I. F. Eissa, Cairo University, Egypt

# DATA CONTROL IN A MAINTENANCE ENVIRONMENT

by

**Michael W. Evans**
**EXPERTWARE, Inc.**
**2685 Marine Way, Suite 1209**
**Mountain View, CA 94043-1125**
**(415) 965-8921**

## ABSTRACT

Significant resources are being committed within industry, by various government agencies, and by the academic community in improving software maintenance techniques. These initiatives are not, however, dealing with the real issues limiting maintenance effectiveness: control of the data configuration released to each user site, the quality implications of a system release, and the test requirements that are unique to the maintenance problem. This paper presents an overview of the different components of software maintenance, as well as the maintenance problems that occur when attempts are made to fix operational deficiencies. An automated maintenance system to handle these problems is recommended. This system, as described in this paper, is geared to handle the different maintenance and control issues and the unique support requirements of both large and small software systems.

Mr. Evans, a Vice President of EXPERTWARE, is a consultant and instructor in software management methodologies, project recovery, and testing and quality assurance. He is author of a series of books published by John Wiley & Sons on principles of productive software management.

## OVERVIEW

In software development, there are many people developing a series of products, each of which builds upon previous products, and all of which are constantly undergoing change by the very nature of the work involved in producing the final system. To make matters worse, a change to one product may require that changes be made to any number of products that currently exist.

During this development, and after release, each version of the software, as well as the engineering and user documentation for each version, must be maintained and supported. All information on related software and hardware products, which versions of the software work with which other versions of software and hardware (Operational Configurations), and which customers have which combination of software and hardware versions (Site Configurations) must also be maintained.

Maintaining and controlling all of the software and software-related data is called **"Software Maintenance"**.

Software maintenance is a frequently misunderstood methodology, to the extent that the control of software during and after customer delivery is often critically mismanaged.

This methodology is a complex process requiring the careful integration of diverse disciplines, technical activities, and administrative project controls. It must address a range of topics which exceed the usual definition of the term, "maintenance".

The methodology can be rigorous, but must be ordered and methodical. Examples of ordered methodologies include, somewhat broadly:

a. Configuration Management. The systematic application of procedures, controls, and management disciplines to coordinate and control all information concerning the software product.

b. Quality Assurance. The evaluation of the form, structure, and compliance of the software product in relation to the data and specifications concerning that product.

c. Structured Design/Management. The systematic approach to creating a design, or modifying an existing design, including ordered application of tools, techniques, and guidelines.

All of these methodologies exist in the software development environment as well as in the software maintenance environment.

The technical disciplines and the maintenance requirements presented in this paper emphasize the control of all data concerning a software product and the necessity to establish a complete and integrated software maintenance environment. This is essential if the integrity of the software product is to be ensured.

## THE SOFTWARE MAINTENANCE ENVIRONMENT

The **Software Maintenance Environment** is composed of the activities and functions necessary to effectively manage and maintain all levels of software products and their related issues. It contains five integrated levels of activities and functions, as discussed below:

1. The top level is the **Maintenance Management and Control** segment of the environment. This level provides the planning necessary to ensure the integrity of the overall software maintenance environment. It defines the tools, techniques and methodologies that are applied to maintaining a set of software products and projects. It also controls the application of resources to the problems of maintenance. This level exerts overall control over the maintenance activity and is responsible for its effectiveness.

2. The second level is the **Data Control** level. This level is responsible for the integrity of all data released to the field, all problems reported in test or operational configurations, and the documentation quality and validity. It contains the integrated set of project procedures, data requirements, quality control practices, and data control practices. These define and control software releases.

3. The third level is responsible for **Software Systems Engineering and Analysis.** This level provides in-plant and field analysis of test systems to evaluate performance and technical attributes of the system and software. It assesses the interface and support characteristics of the software in system and stand-alone environments. The evaluations are conducted in the controlled plant environment and in the uncontrolled user environment. In specific environments, this level is responsible for defining specific site configurations and documenting site-specific data and support requirements.

4. The fourth level is responsible for the **Integration, Testing, and Installation** of software released from the development organizations. This level takes individual software subsystems and integrates them into a complete, executable system configuration, qualifies the system in-plant, and authorizes release to the field. In certain system-maintenance environments, this level is responsible for configuring and testing site systems and installing the systems at user sites. This level is responsible for correcting software problems, providing enhancements, providing modifications due to configurations needs, and providing any additional support.

The coordination and control of the maintenance environment is often more difficult than controlling the development problem. The complexity of managing many, often different, configurations of the same software, the impacts associated with unacceptable system releases and unrealistic customer demands, impossible release schedules, and correcting non-reproducible problems is often overwhelming. The maintenance environment, in spite of these problems, must ensure the technical integrity of all software released to the field, as well as ensuring integrity of the various configurations.

For an effective maintenance environment, each level of activity, and the relationships between them, must be addressed. Solutions must be provided for the administrative and technical requirements associated with each level. The data flow which is expected during software maintenance must be detailed. Overall, there must be a clear identification of the maintenance organization and the interactions between the organizational elements.

The relationships between software maintenance and development must be described and provide a clear hand-off from one function to the next.

## CONFIGURATION MANAGEMENT AND SOFTWARE MAINTENANCE

Making a successful transition from development to maintenance is dependent upon the efficacy of the **Software Configuration Management Process.** It is the link between development and maintenance.

This process encompasses the five levels of the software maintenance environment by providing a systematic approach to storing and controlling the data products, which include the various documents, software, and other data elements that define or specify the different aspects of the software product. The approach used is a **Configuration Management and Maintenance Information System approach.**

Software Maintenance must address the management of software development from the view of the maintainer of software rather than from that of the developer. In many instances, this is the more difficult task. This is due to a variety of situations and considerations, such as:

a. The difficulty inherent in controlling many systems in different field configurations.

b. Dealing with non-reproducible operational failures.

c. Providing timely enhancements to software systems which were developed, all too often, with no consideration to how they would be maintained.

d. Controlling the software and documents (data configuration) released to each user site.

e. The quality implications of a system release

f. Test requirements that are unique to the maintenance problem.

It must be realized that the ultimate success of a software project has a direct relationship to the interaction of software development, software maintenance, awareness of user requirements, and customer support. That interaction is the software configuration management process, as diagrammed in Figure 1.

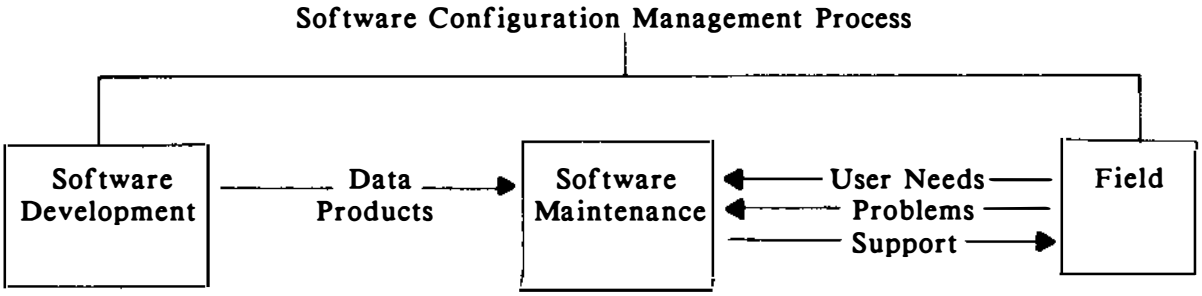



**Figure 1. Software Configuration Management Interaction**

The software-maintenance dilemma is compounded by the fact that software systems are complex. The lack of physical products associated with software development and the complexity of the management program necessary for control of the software maintenance environment also significantly contribute to the maintenance dilemma.

Unless software configuration management disciplines are effective during development, it is difficult, if not impossible, to release an operationally manageable and maintainable product configuration.

In short, if an operationally effective system is to be realized, the software project must plan and manage the disciplines required of software development with a fully integrated, comprehensive, automated configuration management system, not just monitor the progress.

An automated, integrated approach, when followed, provides data control in a maintenance environment. The approach distinguishes between the control of software during development in one area and maintenance control of software in a number of areas. The focus is on the latter, recognizing that most maintenance environments are "after-the-fact" and that structures used to identify data products should contain pointers to maintenance data rather than actual code. If followed, this approach engineers software maintainability into the software life-cycle.

## CONFIGURATION MANAGEMENT APPROACH TO MAINTENANCE

Effective configuration management involves the development, operation, and maintenance of a **Configuration Management and Maintenance Information System (CMMIS).**

The primary function of the CMMIS is to provide information for the planning, control and maintenance of systems and subsystems software.

A key distinction must be made here between "data" and "information". While data can be any unstructured set of facts, information is data that has been made meaningful to the user through some form of organization.

The problem ultimately faced in developing an effective CMMIS is determining the information requirements for making decisions regarding the planning, control, and maintenance of systems.

This understanding is critical to the development and application of an effective software maintenance system. For the CMMIS, information requirements are those that aid in the identification, control, status accounting, audit, review, and data management of given configurations of software.

The CMMIS, like any management information system, consists of large amounts of data. This data is only of value when it can be organized into meaningful packages of information for the user. Clearly, the efficiency and effectiveness of an information system is maximized when the amount of unstructured data is transformed into usable information, and the amount of unusable data is minimized. The creation of meaningful information, and the minimization of extraneous data, is a primary goal in the design or redesign of a CMMIS.

Identification and organization of useful data is based on information requirements determined by user-defined objectives and desired benefits. Once identified, the information requirements are consolidated through the filtering of irrelevant data and the condensing of redundant data and information needs.

At the core of the CMMIS is a database of baseline configurations, specifications, and interface requirements. This information is a formally established set of specifications and data products that describe the result of a specific phase of development. Internal baselines are required of new products as they are developed and are established by the different development groups in an organization.

Proposed changes to baseline configurations are prepared in conjunction with detailed analyses of current configurations, specifications, and interface requirements. The proposed changes are subsequently assessed against current specifications and requirements. Once approved, these configuration changes are entered into the baselines, with traceability always being maintained. The dynamic nature of this database requires responsible, efficient data management.

To understand the implications of a CMMIS, a clear understanding of the concepts embodied by the terms "configuration management", "identification", "control", and "status accounting" is essential. These are defined as follows:

a. Configuration Management is the discipline of organizing and controlling the planning, design, development, and operation of systems and software (separately or in combination) by means of uniform control, identification, and accounting of a product. The goal of these operations is to ensure that the delivered software-related product information meets functional, design, and performance requirements.

b. Configuration Identification refers to a complete description of the physical and functional characteristics of a product, such as, the number of lines of code, processes performed, storage requirements, and performance. This term also applies to technical descriptions required to build, test, operate, and repair a software item.

c. Configuration Control involves the systematic evaluation, coordination, and approval or disapproval of proposed changes to the design and construction of a software item whose configuration has been formally approved internally.

d. Configuration Status Accounting is the recording and reporting of software item descriptions. This includes all departures from the software item (planned or made) through the comparison of authorized design data and the as-developed and test configuration of the item.

The relationships of these major facets of configuration management are shown in Figure 2.

Software Configuration Management includes all of the above items, in addition to specialized requirements for simultaneously controlling the different aspects of software (and firmware), related design, and user documentation. This orchestrated change mechanism is the most frequently misunderstood part of software configuration management. It is however the key element of the CMMIS and is critical to the maintenance of system baselines released to the field.

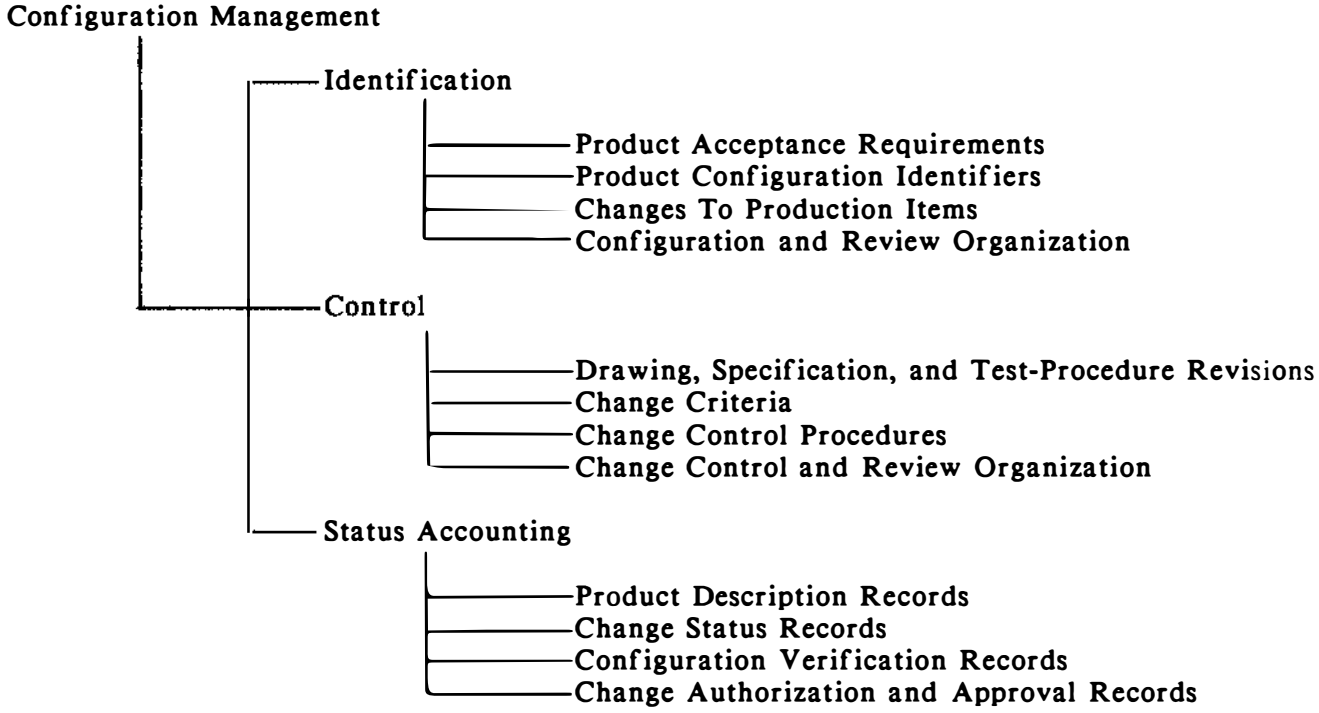



**Figure 2. Major Facets of Configuration Management**

## IDENTIFICATION AND STORAGE OF SOFTWARE CONFIGURATION DATA PRODUCTS

This section describes a generalized approach for identifying and storing software configuration management data.

From a high-level viewpoint, there are two aspects of software configuration management: the management mechanism and the information being managed. Because

of the often extreme complexity of software and associated documentation, software change control and management is best accomplished through automated means. In order to successfully implement an automated software change control and management mechanism, the information to be managed must be clearly identified. This information, referred to as **"Data Products"**, is typically organized into categories.

The types of data products requiring identification and control are those that:

a. Specify plans and procedures for system operation or system development.

b. Specify system functional and operational requirements.

c. Specify subsystem software requirements.

d. Specify the configuration of a product which has been certified to meet subsystem requirements or has been acquired from a vendor for subsequent modification.

e. Describe the use of a software product.

f. Comprise a software configuration that is currently operational.

These data products form a basis for further system development or describe a product which is either operational or is a departure point for system enhancement in a maintenance environment. As such, they are said to comprise formal baselines and are considered to be official products. Baselines, as mentioned earlier, form the essential foundation for automated software configuration management and software maintenance. While internal baselines are not maintained by project management, they are required of the various development groups and must be established and controlled according to established configuration management principles.

In addition to storing, identifying, and controlling changes to formal baseline documents, the CMMIS identifies and tracks the location of all program and project documentation incorporated in maintenance configurations. This includes those documents that are historical and will not be maintained, and documents controlled by contractors' delivery and included in a system release.

Software however, can be comprised of several possible types: a) contractor-supplied off-the-shelf software, which will ultimately be maintained by the project staff; b) newly developed project software which is also maintained by the project staff; c) newly developed software maintained by a contractor.

In order to keep track of the official products described above, the CMMIS maintains a separate inventory list for each major type of item which contains data pertinent to that type of data product. Collectively, these lists comprise the official inventory of formally controlled data products, their locations, and the specific site configuration(s) necessary for software maintenance.

When the information described above is completely identified, an automated software configuration management and maintenance system can be applied. This automated system is typically situated in an area designated as the Maintenance Support Library (MSL). This is the central location for the storage of all information necessary to

identify, control, and account for the statuses of all the maintainable data products. The MSL consists of storage areas for documents, computer media and supplies, and the records necessary to itemize all these data products on request. This information appears centralized from the automated perspective, even though the physical storage areas may be distributed throughout the maintenance organization.

The heart of the MSL is the Development Machine (DM). The DM stores official documents and software products as well as all information concerning products which exist outside the DM, such as hardware, vendor-supplied documents, or vendor-supplied software and equipment. The resident operating system should provide the facilities for storing text and software files in a hierarchical fashion to make identification and access directly relatable to the system being described. It also provides facilities for documentation production and file revision change control through specialized utility tools such as the generic Unix Software Configuration Control System (SCCS), or the DEC VAX Code Management System (CMS).

The file structure which is used to maintain the information in the CMMIS is based on the hierarchical relationships between related data products. The exact structure used for supporting a maintenance project must be based on an assessment of the requirements (ongoing and planned ) which the software must support. These include the amount of data anticipated to be controlled by the system, the frequency of change, and the number of different configurations that the software, when released to the field, must support. The structure must be easily modifiable to support the differing configuration and site requirements. The five major directory structures on the DM which control the different categories of information are as follows:

a. Working Area - All developing software and documentation are contained within this area. The Working Area is where the software is developed prior to release to the field. This area is the most visible window to monitor software development. By requiring internal development organizations to maintain file compatibility, the data can be readily moved, when approved at a project level, to a quality gate, or review.

b. Controlled Program Area - All controlled source software and program inventory lists are contained within this area. The Controlled Program Area contains the structure necessary to provide a version-controlled slot for all maintained source code which has been delivered, thus providing formal control.

c. Configuration Management Area - This is the working area for the MSL Librarian. It contains programs for executing routine information management activities and is used as a temporary storage area for transferring files into the Controlled Program Area.

d. Documentation Area - All approved versions of the software documentation are created here. This includes copies of released documentation, as well as all program reports, action items, and discrepancy reports.

e. Test Area - Newly generated executable object code to be used in testing is located in this area.

These areas, or directories, are located under the root directory, as shown in Figure 3.

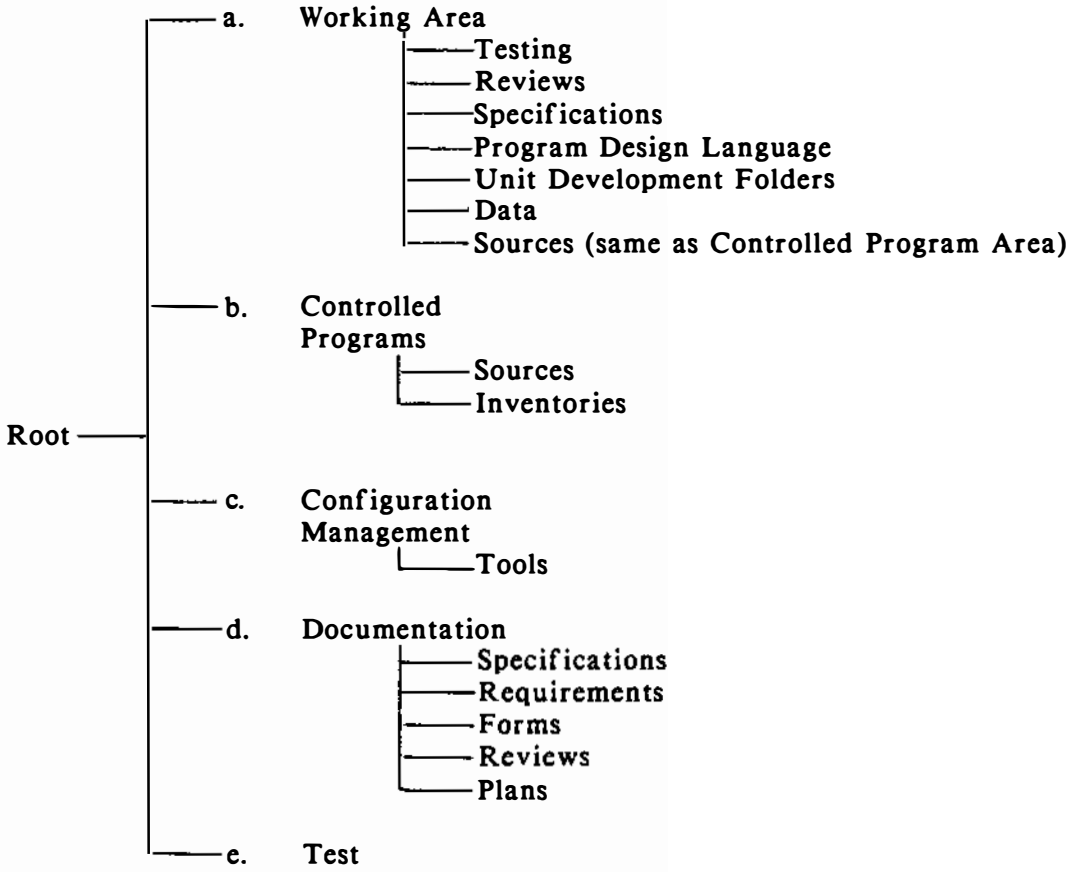



**Figure 3. Configuration Management Directory Structure**

The MSL Librarian controls passwords which allow only designated MSL staff members to change the contents of areas (b) through (e) above, while any project member may read or copy the contents. The Working Area content is not necessarily under MSL control, but the MSL defines and maintains its structure.

## AUTOMATED SOFTWARE CONFIGURATION MANAGEMENT DEVELOPMENT AND SUPPORT TOOLS

EXPERTWARE provides the following computer-based tools that are used to support a CMMIS. Specifically, the CMMIS consists of the directory structures described earlier, databases for SPRs and configuration status accounting data, and utilities for easing the tasks of data product collection, analysis, and publication or reproduction.

EXPERTWARE's Configuration Management Toolkit (CMT) and Documentation Support Toolkit (DST) provide seven integrated automated tools in the following areas to support the configuration management and the maintenance environment described in this paper:

1. **Configuration File Management**

The EXPERTWARE CMT Configuration Control Tool provides for an integrated method of allocation and control of files for data products and their transfer between controlled and working areas of the development machine. The user need only designate the location of the topmost level of the system component to be moved or controlled and the tool automatically moves each file in the structure and maintains the hierarchical structure integrity. The tool allows the user to share common software and documentation across multiple configurations and versions of the product, thus requiring less system data storage area.

2. **Problem Reporting**

The EXPERTWARE CMT System Problem Report Tool allows the online input (by anyone) of System Problem Reports (SPRs) and their updating and maintenance in a Program Support Library (PSL)-controlled database. This tool provides control of problem reports, generation of SPR agendas and summary reports for control board meetings, and production of reports showing the current and historical status of SPRs.

3. **Library Cross References**

The EXPERTWARE CMT Cross Reference Tool automatically identifies and records the references to or from the elements in a product component. The reference lists help designers, engineers, and programmers determine how a proposed change may impact the existing configuration.

4. **Automated Make/Build Tools**

The EXPERTWARE CMT Build Support Tool automates the process of generating, compiling, linking, and building subsystem software in an executable form. This tool operates interactively to allow a PSL operator to override defaults and specify alternate source library input. The object and executable images are routed to pre-specified PSL files established for them.

5. **Version Description Document**

The EXPERTWARE CMT Version Description Tool allows the user to specify the exact combination of system components to be used to generate a detailed report on the contents and changes of one or more versions of the system. All of the information needed for a Version Description Document (VDD) is retrieved automatically by this tool from the files contained in the PSL controlled area. The VDD details the contents of a build of a subsystem or system release, shows each module by release level that is included in the build, all SPRs closed by this release, and any discrepancies remaining open. This tool also formats the data and generates the basic content of the VDD. The formal VDD is then completed by PSL staff using one or more of the tools in the EXPERTWARE Documentation Support Toolkit (DST).

6. **Documentation Support**

The EXPERTWARE DST Document Generation Tool augments the standard system text editors and formatters, or acquired word processing systems, by automatically collecting the specification data files into new files and formatting these into designated system

and subsystem specification formats. This provides a capability for semi-automatic generation of formal documents from PSL files. This tool additionally provides commands for the generation of tables of contents, indexes, and all formatting normally required for documentation generation.

7. **Standards Templates**

The EXPERTWARE DST Template Definition and Generation Tool provides for the automated implementation of the project standards. These tools allow a user to select a particular template for an individual data item. Using the template, the data skeleton is filled out at the terminal and the completed data template is either returned as a file to the directory structure from which it was requested, or to a user-specified directory. The use of templates standardizes the data-collection process and significantly improves data collection quality.

## SUMMARY

This paper has provided a discussion of the software maintenance problem, pitfalls typically encountered and recommendations for correcting the problem.

The paper has also discussed the relationship of data control between the development and maintenance phases of a software project by discussing the importance of software configuration management - the central, all-encompassing thread which links the various aspects of software development together. This link, and the automated tools necessary to manage software maintenance data, have been characterized as a Configuration Management and Maintenance Information System (CMMIS). The CMMIS is an integrated set of automated tools, data files, and procedures which, when implemented, provide a complete environment for maintaining software. The EXPERTWARE Configuration Management Toolkit (CMT) and Documentation Support Toolkit (DST) products provide an interface which allows the non-programming MSL staff to enter, retrieve and manipulate the files in the MSL, as well as providing a "non-programming" user an easy-to-understand interface to MSL software. The MSL and the capabilities provided by the CMMIS are the link that the maintenance organization has to the development organizations. It is the means by which software transitions to an operational status and is the vehicle by which site configuration are maintained.

## GLOSSARY

| | |
|---|---|
| Baseline | A formally established and approved set of specifications and data products that describe the result of a specific phase of development. |
| Baseline Configuration | See "Baseline". |
| Configuration | The assembly of software and/or hardware components and data products that interact compatibility to form a complete system. See "Baseline Configuration", "Data Configuration" and "Maintenance Configuration". |

| | |
|---|---|
| Configuration Management | The systematic application of procedures, controls, and management disciplines to coordinate and control data and manage the data configuration and releases. |
| CMMIS | Configuration Management and Maintenance Information System. |
| CMT | EXPERTWARE's Configuration Management Toolkit. |
| Data Configuration | All the product-related software and documentation released to the user. |
| Data Products | The documents and software having to do with the management of the software development process and the transition from one development phase to another. |
| Development Environment | The operations and functions necessary in developing a software product. |
| DM | Development Machine. As the heart of the Maintenance Support Library, the DM stores official documents and software products, as well as information concerning products outside of the DM. |
| DST | EXPERTWARE's Documentation Support Toolkit. |
| Maintenance | See "Software Maintenance". |
| Maintenance Configuration | All of the software and/or hardware and documentation that comprises the system under maintenance. |
| Maintenance Environment | The operations and functions necessary to effectively manage and maintain all levels of software products and their related issues. |
| Maintenance Support Library | The central location for the storage of all information necessary to identify, control, and account for the statuses of all the maintainable data products. |
| MIS | Maintenance Information System. |
| MSL | See "Maintenance Support Library". |
| Program Support Library | The organization performing configuration control tasks and configuration status accounting. |
| PSL | See "Program Support Library". |
| Quality Gate | A formal or informal evaluation of a product's complience to technical content or standards. This is conducted at the completion of development of the product and before general release. |

| | |
|---|---|
| Site | The customer's or user's installation. |
| Site Configuration | The system of interacting software and hardware products found at the user's site (see "Configuration"). |
| Software Maintenance | The tracking, manipulation, and control of all of the software and software-related data. |
| SPR | System Problem Report. A formal means to document and submit problems, questions, issues, or concerns related to the product. |
| VDD | Version Description Document. |

## REFERENCES


Berghoff, et al.; Software Configuration Management - An Investment in Product Integrity, Prentice Hall, Englewood Cliffs, NJ, 1980.

Dean, William A,; "Why Worry about Configuration Management", Defense Systems Management, Vol. 2, No. 3, summer 1979.

Evans, M.; Productive Software Test Management, Wiley and Sons, New York, NY, 1984, pp. 66-82.

Evans, M., et al.; Principles of Productive Software Management, Wiley.and Sons, New York, NY, 1983, pp. 31-54.

Evans, M., Marciniak, J., Software Quality Assurance and Management, Wiley and Sons, New York, NY, 1986.

McCarthy, Rita; "Applying the Technique of Configuration Management to Software", Quality Progress; Oct. 1975.

# PROPOSAL FOR A SOFTWARE DESIGN CONTROL SYSTEM (DCS)

Robert Babb II
Dick Hamlet

Department of Computer Science and Engineering
Oregon Graduate Center
Beaverton, OR 97006

**Abstract**

It is proposed to extend the idea of source-code control or revision control to the entire software development cycle. The extension, called a design-control system (DCS), provides two services for development:

1) A DCS allows safe cooperation among the members of a design team. Software components are checked out and back in with syntactic and semantic checks for integrity of code and documentation, so that changes are less likely to turn a working system into a nonworking one.
2) A DCS aids in making changes by keeping track of their ramifications throughout the whole design, prompting its human user to examine and update each part of the developing system when any connected part is altered.

A DCS is intended to help its users cope with the complexity of large systems during development and maintenance. It is particularly applicable to realtime programs, system software that exists in multiple versions, and supercomputer scientific programs.

**Biographical Sketches**

Robert Babb is an Associate Professor at OGC. His primary research interest is software engineering environments for scientific parallel processing. His large-grain data flow (LGDF) model of parallel computation provides a way to describe and control the development of complex concurrent software. He received his Ph.D. from the University of New Mexico in 1974.

Dick Hamlet is a Professor at OGC. His research interests are theory of software engineering (particularly software testing), programming languages, and computer-aided text processing. He received his Ph.D. from the University of Washington in 1971.

## 1. INTRODUCTION

Revision-control systems such as SCCS [1] and RCS [2, 3] are a good model for managing the entire development cycle for a piece of software. Existing tools handle files containing source code, and provide for the automatic creation of a system from its pieces, using configuration and time-stamp information to get the latest version of the proper modules (using, for example, the **make** program [4]). They also support the process of code modification by controlling access to source modules. This narrow focus can be broadened to cover the complete development cycle: requirements, specification, design, code, tests, and maintenance history. Just as source code can be controlled and checked, so can changes to any part of the information complex be monitored.

Source control for code includes the automatic construction of a consistent, up-to-date system. In development control, interactive evaluation of changes is more appropriate, because the ramifications of change are hard to predict. Since no algorithm can automatically correct for arbitrary ill-considered changes, the human user must be informed of potential problems before changes are made, then aided in keeping track of solutions as they occur. In the dialogue between a human developer and the machine, almost all the intelligence rests with the former. The computer aids must use simple syntactic indicators of change to suggest problem areas that the developer should examine.

However, automating the module integration process may not be the most important function of revision control. Changes are made by people, perhaps many people working together. It is better to aid them throughout program design, coding, testing, and maintenance, than to attempt design repair after a modified system fails. Programmers have a strong desire to do their work well, but are often defeated by the complexity of the system being developed. The aid we envision would enhance communication between different programmers, but should also be invaluable to an individual working alone on a program.

Concurrent systems, in which the developed program will actually execute in parallel, are historically the most difficult to build, and the most in need of support. Concurrency introduces a natural division of modules corresponding to subsystems that will be executed in parallel; the payment is the additional complication that results from synchronization and message passing between subsystems. When the program being developed depends for its performance on the details of the hardware configuration (as numerical software often does), there is the additional difficulty of keeping a base source and versions tailored to a number of machines.

Systems programs do not always involve concurrency, but as general-purpose tools they exist in many interconnected versions and have long lives marked by revision and enhancement. Their developers and maintainers thus must handle the information explosion created by simultaneous changes along time and version dimensions.

## 2. EXTENDED REVISION CONTROL

The design control system (DCS) we envision focuses on source code and on surrounding phases of development that support coding (specification, design, testing). Under its control are units (modules) that can be assembled into a running software system. Programmers should be able to work on any unit in safety, knowing that a modified test system can be constructed without losing the older "current best" system by accident. Furthermore, programmers should know that there will be no unexpected problems caused by multiple simultaneous updates of the same or distinct source components by different people. And during the modification process, there should be global awareness of the scope and impact of any proposed change, taking advantage of all available machine-encoded information.

### 2.1 Components of a System under Development

In support of the stored source code, the following are the components of a DCS:

*System Requirements.* The requirements document describing software is usually in English, and usually not a model of precision and clarity. However, it is the basis for specifications and designs that begin the development process, and it can be stored with links to the elements that arise from it.

*Module Specifications.* Natural-language specification is also the rule in large systems, but for individual modules more precise techniques are available. A module implementing an abstract data type can be described by axioms in a way that allows testing [5], and some logic specifications can also be checked by tests.

*System Design.* Designs can be expressed in pseudocode, using a program design language (PDL). Since these languages are very like the programming languages used to code the designs, it is easy to establish good links between these levels. For special kinds of software, special design techniques are appropriate. For example, the LGDF technique [6] can be used in concurrent applications.

*Module Interface and Interconnection Definitions.* Managing the data aspects of interaction between modules involves, for example, ensuring type agreement on communication paths (parameters for subroutines, message channels for concurrent processes). Some of the control aspects of the system design can also be captured by explicitly representing potential use relationships among modules (a software "wirelist"). These interface and interconnection definitions can be kept up to date automatically.

*Unit- and Integration-test data.* Interface definitions are static, depending only on the program structure expressed in its syntax. To make use of any other information requires the program semantics. Test data is the most practical form stored semantic information could take. Unit tests can be linked syntactically with specific data interfaces, and semantically with module specifications. Integration tests are judged against the system's specification as a whole. In addition, tests may be required to meet coverage criteria of some kind.

*Semantic constraints.* Each kind of software system has special features that can serve as an indicator of its quality in addition to test data. For example, in systems intended for parallel execution some concurrent behaviors are allowed while others are erroneous. This information can be expressed in terms that may be verified — for example as grammatical descriptions of permitted and forbidden execution traces for sets of test points.

### 2.2 Properties to be Preserved by Revisions

Maximum help is provided to a programmer seeking to make a change if its potential effects are explicitly displayed. This is quite different from later checking for potential flaws caused by changes. For example, if a programmer were aware that a proposed change affects almost every interface in the system and invalidates all unit test data, that person might think twice about making it.

Any change begins with the editing of a component stored within the DCS. The potential implications of the change can be estimated from the syntactic relationships among components:

*Changing a specification* implies that design and code for the specified module will be changed and its test data will be altered or extended. There is the potential for interface changes, and other modules may have to change even if the interface does not.

*Changing an interface* requires changes to modules and their designs (those that define it, and those that make use of it), and changes to tests using that interface.

*Changing code* may require changes in all the other stored information supporting that code.

*Changing semantic constraints* may require changes to the module communication structure or may involve only specification or code changes.

*Changing tests* may affect all other stored information, but in a nearly complete development may influence only code.

In each case the potential implications of change are usually more extensive than the actual ones. The programmer who intends to edit a component can be shown the potential for change, but must prune this to reflect reality.

In the traditional "batch" mode for system revisions, when a change is made its implications for other components are not investigated. Later, when all editing is completed, a batch of such changes are incorporated, with some syntactic diagnosis for system integrity. In the interactive mode proposed here, the checks would be applied immediatedly as each change is incorporated. This is possible because widely scattered syntactic sites of change can be located using additional design information. For example, the module interconnection description can be used to predict the implications of a parameter change in a code subroutine.

Each actual change can thus prompt for associated changes necessary to maintain consistency of specification, design, interfaces, code, and test data. When the syntax of changes is consistent, the modified system can be executed using stored test data to determine partial consistency of specification and code, and partial satisfaction of semantic constraints.

### 2.3 The Modification Process

The features described above are all intended to help the individual programmer, not to control multiple simultaneous access to system components. A DCS must also provide facilities to help people work together. The mechanism for this is a "check-out/check-in" scheme based on potential propagation of changes.

Each component that might be edited has a potential sphere of influence that extends in two directions: across other components of the same kind; and, into components of different kinds. For example, changing code module *M* may have an impact on other code modules; it also touches the design and test data for *M*. A DCS can calculate the worst-case ramifications of any proposed change. The person making the change can then be asked to select from the list of possible influences those that are likely. These components are then checked-out for change, in a "sheltered workspace" so that the existing system is not compromised. Others may not specify for check-out any components already checked out. As changes are made and the extent of interactions becomes clearer, the check-out list might need to grow. Should the needed component be checked out elsewhere, a potential for deadlock exists, but in any case the people involved should immediately get in communication, and a DCS can insist that their intentions be resolved before editing continues.

A DCS can calculate, for the changes made by one person, all secondary information that it possessed for the unmodified system. This information includes test results, the module interconnection pattern, and parallel execution sequences realized in testing. Unlike syntactic consistency, which can profitably be checked interactively, this secondary information is semantic, dependent on the existence of a complete, executable system, and so easily checked only when changes are complete. Test results must meet specifications and semantic constraints must be observed.

When a change is complete, all checked out components are again checked-in and the sheltered workspace cleared. When several people are working together, it can happen that two syntactically independent changes (hence ones in which no conflicts arose in check-out) may nevertheless conflict semantically at the time the second person tries to check-in. For example, an integration test point may fail because two modules were changed by different people, the first-checked-in change is really to blame, but the problem does not appear until the second check-in. Such a situation is obviously nasty, but the people involved can investigate it by looking at the history of modification since check-out of the component that failed to check in.

It should be noted that nothing in this description of "changes" precludes application of a DCS to that drastic change from design to coded system that is called "initial development."

### 2.4 A Simple Example

Imagine that the information stored about a mathematical function module `MathFun` includes requirements (in English), a design (in pseudocode), a logic specification, source code, unit-test data, and a maintenance history. A programmer decides to give `MathFun` an additional parameter to let it handle additional cases, and to eliminate another routine `OldFun` that formerly handled these cases. When the programmer attempts to alter the code modules for `MathFun` and `OldFun`, the DCS can search the design for these names. It can follow links between design and requirements to provide the reasons why the routines were originally introduced, and it can supply a maintenance history for them. The proposed change requires that modules invoking `MathFun` or `OldFun` be changed, but probably not that *their* designs be altered. However, the specification and unit test data for `MathFun` must now include the extra cases (probably by modifying and incorporating the similar information from `OldFun`). Knowing that the DCS will insist that all these changes be made together, or that none be made, the programmer might well reconsider.

Suppose that the programmer persists, however. Then the DCS will supply the list of potentially influenced components for checkout. The programmer might decide not to check out designs and specifications outside `MathFun` and `OldFun`, assuming that these need not be changed. As the changes are made, the DCS can prompt for each associated change, if it cannot be made automatically. For example, the actual alteration of the parameter list in the `MathFun` code module would call up its unit-test module for change, plus the corresponding design and specification modules, and the code modules of routines calling `MathFun`. As the latter are changed, their designs might require updating, and since these were not checked out, they would be added to the checkout list. The programmer would probably refuse the opportunity to check out and modify unit test data for the calling routines, since that is unlikely to need alteration. The maintenance history and the interface descriptions can be automatically updated. Finally, when all the prompted changes have been made (or the programmer has refused them as unnecessary), the complete unit tests can be applied. If failures are found, checkin cannot take place until they are repaired. While work is in progress, another programmer attempting to work on (say) `OldFun` will be refused access.

## 3. APPLICATIONS AND THEIR SEMANTIC CONSTRAINTS

Any complex piece of software can benefit from a DCS that tracks changes among its components. Most features of a DCS are common to all systems being developed, but semantic constraints differ in different applications. The semantic constraints, like test data, are used to gauge integrity of a change being checked in. Where a constraint can be given that addresses the essential complexity of a system, it can be invaluable in detecting mistakes in the

development process. Tests probe only the functional part of semantics, to which specifications are largely devoted. Other aspects of behavior, notably performance, are more difficult to describe. Here a DCS has the advantage that when a change is made, an old, best-working system exists. At checkin time for a modification, the existing system can be used to generate nonfunctional constraints for the new version to satisfy.

*Concurrent systems.* A major source of complexity in software for multiple processors executing in parallel is the synchronization of communication between processors. The appropriate semantic constraints therefore involve these communications. First, we intend to use the large-grain dataflow (LGDF) model of parallel computation [6]. In the LGDF model, the overall computation is given as a collection of processes linked by data paths. Processes are scheduled by the dataflow rule: a process can be activated only when all required inputs are available, and all outputs have been absorbed. This rule immediately eliminates many incorrect interleavings of execution sequences among the processors, and can be checked using test data. Furthermore, additional constraints may be placed on the interleaved executions, for example in the form of required or forbidden sequences of process activations, and these too can be checked. The combination of dataflow rules with other sequencing constraints gives us some control over the very large space of possible interleavings. Within the constraints it may prove possible to vary the relative processor speeds and attain some kind of "coverage," similar to conventional test coverage, of the parallel-execution space. The LGDF model is also a good formalism for design itself, providing solid links between design and code, and design and test.

*Systems programs.* The complexity of general-purpose system software (e.g., compilers, operating systems, utility programs) arises from multiple system configurations and from interactions between what are apparently independent programs. For example, the same compiler may have a subset and a full-language version; an operating system for the high-end machine in a family will be different from the same system at the low end. When an editor is changed, it must not produce files that cannot be handled as input by a compiler; operating-system services needed for an existing library cannot be changed or removed. The problem of interaction among programs can be approached using test data. For example, when an editor is being checked in, its test output can be compiled. Multiversion software requires the constraint that all versions agree on each test, and each remains within the restrictions of its configuration.

*Scientific software.* Traditionally, scientific software is the best understood, the easiest to test, and the least subject to modification. However, scientific codes have always been the first to need faster, larger processors. The multitude of competing supercomputer designs complicates the task of the scientific programmer who needs their power. A single program may have to be reworked for each different machine, and the details of these versions are critical in achieving good performance. Yet the program retains a single algorithm, and there is a need to manage its design and maintenance for all machines at the same time. A DCS can meet this need, using special semantic constraints. In addition to the dataflow and concurrency constraints, measures like the degree of vectorization are appropriate. When a module is checked in, the DCS can verify that its performance on test data has not been substantially impaired. Furthermore, since supercomputers vary greatly in cost and capabilities, changes in the parts of an algorithm common to all machines should preserve their ranking—the code should continue to run faster on faster machines.

## 4. IMPLEMENTATION

Most software-engineering environments have a database foundation, since placing development information in this form makes it easy to record changes, to make links between the products of different phases, and to process queries. However, conventional databases and

their data models are not well suited to the large records (a section of a requirements document, for example) and long-duration transactions (the check-out to check-in time for a module update could be days, for example) of a DCS. Furthermore, use of a database multiplies the overhead of building a prototype DCS for experimentation. For these reasons we plan to use unstructured files for component storage. This decision is supported by our use of the UNIX™ system, whose existing programs and prototyping methods use unstructured files. We plan to use UNIX tools whenever possible; as an obvious example, **diff** and **grep** can be used to compare an updated file with an existing version and find patterns in the changes.

In a prototype, most of the cross referencing between files containing source programs, specifications, designs, etc. can be accomplished by searching the text files for common identifiers. Where there are no intrinsic identifiers, as in files of test data, names linking the data to other files can be added. If searching is too slow, cross references between files can be made more efficient by constructing a file of linked positions, blocking the files to prevent changes from invalidating too many links.

The basic check-out/check-in mechanism is available in existing revision-control software; it can be extended to include testing by conventional preprocessor techniques. For example, to require that a changed module reproduce the test results from an earlier version, it can be surrounded by a driver that feeds it the regression test data, directs the output into a temporary file, and then compares this with stored results. The creation of the test harness, compilation and execution of the modules being checked in, and reporting failures in the outcome, can all be arranged using UNIX **csh** scripts.

## References


1. Marc J. Rochkind, The Source Code Control System, *IEEE Transactions on Software Engineering,* Vol. SE-1, No. 4, Dec., 1975, pp. 364-370.
2. Walter F. Tichy, Design, Implementation, and Evaluation of a Revision Control System, in *Proc. 6th Int. Conf. on Software Engineering,* Tokyo, Japan, Sept. 1982, pp. 58-67.
3. Alan L. Glasser, The Evolution of a Source Code Control System, *Software Engineering Notes,* Vol. 3, No. 5, Nov. 1978, pp. 122-125.
4. Stuart I. Feldman, Make - A Program for Maintaining Computer Programs, *Software Practice and Experience,* Vol. 9, No. 4, April 1979, pp. 255-265.
5. John Gannon, Paul McMullin, and Richard Hamlet, Data abstraction implementation, specification, and testing, *TOPLAS,* Vol. 3, July 1981, pp. 211-223.
6. R. G. Babb II, Parallel Processing with Large-Grain Data Flow Techniques, *Computer,* Vol. 17, No. 7, July 1984, pp. 55-61.

# A Programmer's Database for Reusable Modules

T.G. Lewis, Oregon State University
I.F. Eissa, Cairo University, Egypt

T.G. Lewis is Professor of Computer Science, Oregon State University, Corvallis, OR. Lewis is active in IEEE-CS (Reviews Editor of IEEE Software Magazine), author of 20 books and 35 technical papers, and one of the founders of the NW Software Quality Conference. His current interests are in software tools, and computer-aided programming.

I.F. Eissa is Associate Professor of Computer Science at Cairo University, Egypt. He is currently visiting Oregon State University, Corvallis, OR. He has a B.Sc. in mathematics from Cairo University in 1968, and a Ph.D. in Computer Science from Manchester University, England in 1976. His research interests are in Programming Languages and Database Management Systems.

## Abstract

It is now widely accepted that current methods for software are not totally satisfactory. The evidence for this is all too familiar and ranges from error-ridden programs to complete project failures. A variety of tools and techniques have been proposed and developed which attempt to alleviate the situation. The result has been new programming languages, using visual programming, developing knowledge-based assistance for software engineering, software reusability, and various other efforts. In this paper we describe a tool which assists a programmer in reusing existing source program modules.

A reusable module is any collection of procedures, functions, programs and their associated attributes (such as documentation) that can be used or modified in order to be used as a component in another program. While there are many program libraries used in practice, they do not embrace the larger issue of reusability. An essential factor in this larger issue is how components are identified and specified. How should components be catalogued? How should interconnections and dependencies among modules be made visible to a user?

This paper describes a method of storing and then finding reusable code modules in a special purpose Program DataBase (PDB), called ***GrabBag***. GrabBag has been designed and implemented to endorse the larger issue of reusability. Users search GrabBag in an exploratory way that does not assume any knowledge about GrabBag's environment using an automatic backtracking mechanism and alternate search paths.

# 1 Introduction

It is now widely accepted that current methods for producing software are not totally satisfactory. The evidence for this is all too familiar and ranges from error-ridden programs to complete project failures. A variety of tools and techniques have been proposed and developed which attempt to alleviate the situation. the result has been new programming Languages [1], using visual programming [2], developing knowledge-based assistance for software engineering [3], software reusabilty [4], and various othner efforts [5,6,7]. In this paper we describe a tool which assists a programmer in reusing existing source program modules.

A reusable module is any collection of procedures, functions, programs and their associated attributes (such as documentation) that can be used or modified in order to be used as a components in another program. Working on reusability has been advocated as a means for increasing programmer productivity by many authors; Jones [8], Standish [9], Horowitz [10], and Neighbors [11]. They emphasize that great care and concern should be taken when identifying components, methods of specifying components, the form of the components, and how the components are to be cataloged. Also, because many of the analysis, design, and implementation decisions are absent from the code itself, there should be some attention given to the notion of reusable designs as well as reusable implementation.

As an example, UNIX Source Code System [12,13] is a very successful system used extensively in practice to store changes in source code over a period of time. SCCS is valuable for version control, but it does not address many other issues of reusability. For example, SCCS stores shared modules in a redundant way [14], and it does not support the detection, collection, and dissemination of information regarding the interconnection and dependencies among modules.

This paper describes a method of storing and then finding reusable code modules in a special purpose Program DataBase (PDB), called GrabBag.

GrabBag differs from traditional program libraries in a number of important ways:

1. GrabBag stores source code text along with associated document, text, interface specification, and test files. These files are called attributes of the module and contribute to the reusability of the modules.

2. GrabBag's underlying structure is a restricted form of a Network Database Management System that Comprises many features designed for convenient access of resuable modules. These features include:

   (a) The retrieval mechanism allows browsing: exploratory searching which does not assume any knowledge about GrabBag's environment.

(b) Because browsing is characterised by frequent failures [15], GrabBag provides a mechanism for automatic backtracking, and alternate search paths.

(c) The user interface is menu-driven: Menu-based systems are among the most user friendly interface systems [16].

(d) The menus are alterable by the user so that changes in the database are reflected as changes in the hierarchy of menus.

(e) Modules are indexed according to their use rather than by their name. Hence, modules are stored and retrieved according to a network of categories. These categories are not fixed in advance, instead they are defined by GrabBag builders.

3. GrabBag is language independent, hence modules from any programming language may be catalogued and retrieved through GrabBag.

After a programmer has defined a high level view of the system or application to be developed and has some idea of the major code segments that will be required, the programmer may use **GRABBAG** to find reusable modules, if they exist. Conversely, as new code is produced, modules that may be useful in the future can be added to those already stored in **GRABBAG**. Thus, there are two groups of programmers that use **GRABBAG**: searchers and builders.

Searchers look for a module that hopefully has been previously added to **GRABBAG**. A searcher selects the item **LOOKUP MODULE** operation to find and copy existing modules for reuse. Searchers cannot make any changes to any part of **GRABBAG**, but they can see what is in it and get copies for private use.

Builders form a much smaller group who put modules into **GRABBAG** and define access paths to newly entered modules. Builders use operation **ADD MODULE**; which is protected by a password.

# 2 MODULES AND ATTRIBUTES:

Reusability implies more than repeated use of program fragments or simple program objects such as abstract data types. To achieve reusability documentation, error messages, I/O formats, help messages, graphics, and test data should be separated into reusable attributes. Thus in **GRABBAG**, a module is defined as a collection of text files containing one distinct attribute per file. The attributes defined in **GRABBAG** are identified by file name extensions as shown below.

.DEF the interface definition;

.DOC the module documentation;

.ERR the error messages;

.FRM formats used by the module;

.HLP for help prompts;

.IMP for the source code;

.INT interface specification.

.NEW for development of history of module;

.PIC any icons or pictures used by the module;

.PRM prompts used;

.RES resource files for the module;

.TST test data

The three-letter extenders identify the contents of an attribute file. Every module consists of one or more attribute file, each of which can be reused, separately or together. The consistent naming convention of these extenders is extremely useful to a designer because it allows anyone to retrieve and display all interfaces, all text or graphics, and all program logic embedded within a large system. For example, all interfaces can be obtained by printing all INT files embedded within a certain program.

The **DEF** attribute file contains the interface specifications for the module. It corresponds to the compilable **DEF MODULE** text required by every Modula-2 module. The **INT** attribute is similiar to the **DEF MODULE**, but the **INT** may contain additional interface specificiation information that is of value to the designer and maintainer of a system, even though **INT** may not be compilable.

Similarly, the **DOC** attribute file contains useful design and maintenance information that is not compilable. Instead, the **DOC** file accompanies the reusable module, and can be reused, itself. Reusing the same, or similar documentation, saves implementation time.

The **NEW** file is related to the **DOC** file: it contains a running log of programmer notes on bugs and changes to the module. The **NEW** file gives useful historical information concerning the modifications and uses of the module over a long period of time.

It is useful to separate program logic from text as well. Therefore, error messages (ERR), I/O formats (FRM), help text (HLP), and optional graphics images (PIC) are placed in separate attribute files. The separation of text, graphics, and program logic vastly improves the lot of the maintenance programmer, and makes it rather easy for any other programmer to adapt a reusable module to another purpose.

Other attribute files may be included with each module stored in **GRABBAG**. For example, test data (TST) may be retained from the development phase in order to be reused when the module is modified and re-tested. Other information such as system configuration data (RES) and module re-configuration prompts (PRM) are included, but

their discussion is beyond the scope of this paper.

Reusable modules that are designed for ease of reuse and maintainability are called M-Objects (Maintainable Objects). They can be thought of as an extension to the notion of an abstract data type. Reuse is equivalent to instantiation of an abstract data type, and separation of the attributes of a module is merely a mechanism to extend the notion of object inheritance.

**GRABBAG** permits users to select a module's attributes one at a time, view or copy them, and freely add new attribute files. This capability radically alters the way in which new systems are constructed and perhaps more importantly, the way they are maintained.

# 3 THE PROGRAM DATABASE

**GRABBAG** contains sets of option lists that allow the searcher to successively refine the description of the module desired. An option list is a set of options (called categories); each category is a text prompt used to lead the searcher to a desired module. A series of indivual categories that lead to a module is called an **access path.**

There may be many different ways to describe a module, hence there may be many paths leading to each module. **An alias** exists when there are multiple access paths to the same module. These different paths form a network of access paths from the root option list to each module stored in the database. Consequently, **GRABBAG** is a restricted network database management system.

A searcher may wish to find several different modules stored in **GRABBAG**, so the system can be instructed to remember a point in the path where a diverging search path is begun. Later, the system will automatically backtrack and resume the search along alternate paths.

**GRABBAG** forms a directed graph where each node in the graph corresponds to either an option list or a module. An arc in the graph represents a single category in an option list. The graph G consists of n nodes and m arcs:

G = {P,E,i}

where P = {pi | i=1..n} nodes, and $p_{root}$=root node,

E = {ei, | i=1..m} arcs, and

i = connectivity

An access path A is a sequence of nodes traversed from the root node to any other node in G,

$$A = P_{root} - - > Pj - - > .. - - > P_k.$$

Access paths may have nodes in common, and there may be more than one access path between two nodes $P_{root}, P_j$.

i is determined by the set of categories which connect $P_i - - > P_j$ for every path A.

In the graph G, modules are always terminal nodes with no outgoing arcs, and option lists are always nonterminal nodes. The title of a node is the category of the incoming arc of that node. If two or more arcs connect to a node, the one used by the current access path is the **primary path** and the others are called **aliases.**

Figure 1 illustrates the relationship between option list nodes, module nodes, and a special token called the mark. The **root** node $P_{root}$ is the logical start or center of the graph maintained by **GRABBAG.** The root has one outgoing arc with the category 'First Option List'. Figure 2 is a text example for Figure 1. This text example will be used through the paper for illustration purposes.

**Mark** is a token which resides in one node of G. The location of mark can be changed by builders, but there is always exactly one mark. Mark is used to mark and "remember" a node that will be linked with some other node, later on. The marked node is the target, or destination of a connecting link in the network, G. An * is used to designate the currently marked node:

$$P_i - > P_j*$$

Data Structures for the access paths are as follows. The nodes of G have four fields: NAME, TYPE, NEXT, AND ATTACHED, (Figure 3). Name is an index into the random access text file for the database where the text of the categories is stored. TYPE is a character indicating whether the node is either a module node or an option list node. **NEXT** and **ATTACHED** are pointers to other nodes.

The code and attributes for a module are stored in a group of files that have the same base name and extenders like '.DOC' to identify the different attributes of the module. The base name is stored in a node which has a NIL **ATTACHED** pointer and has its **NEXT** pointer to a node which contains the extenders of the attribute files associated with the module. Module attributes can be added after the initial entry into the system.

Nonterminal nodes are the option list nodes that are used to guide the searcher to the module nodes. Categories point to either a module node or an option list node. Those

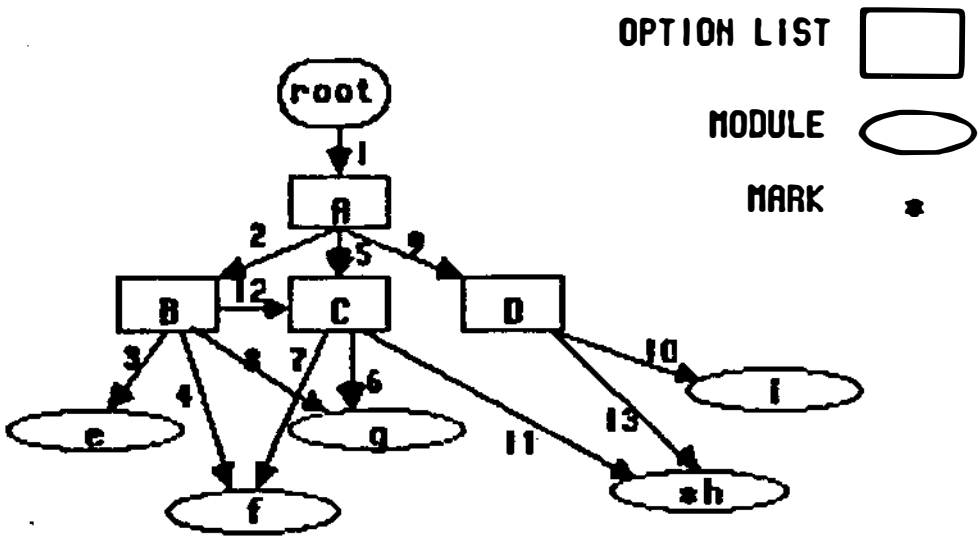



Figure 1. Example of *Grabbag* Network Graph G.

---

1. First Option List
2. Math Functions
3. Set Operations
4. Matrix Operations
5. Physics Applications
6. EE Circuit Calculations
7. Partial DifEq Solutions
8. Complex Number Functions
9. Statistics Packages
10. Sensitivity Analysis
11. Curve Fitting Approximations
12. Uses in Physics
13. Averaging Unreliable Datapoints

e. .DOC.DEF.IMP.FAM
f. .DEF.IMP.DOC.TXT.TST
g. .DOC.DEF.IMP.HLP
h. .IMP.DOC.DEF.RES.FAM
i. .DEF.IMP.DOC

Figure 2a. Category and Module names example for Figure 1.

A First Option List
Math Functions
Physics Application
Statistics Packages

B Math Functions (access path A B)
Set Operations
Matrix Operations
Complex Number Functions
Uses in Physics

C Uses in Physics (access path A B C)
EE Circuit Calculations
Partial DifEq Solutions
Curve Fitting Approximations

Figure 2b Examples of option lists (from Figures 1 and 2a)

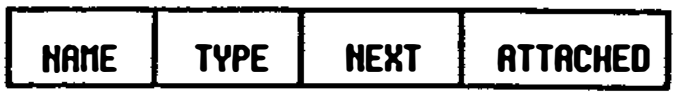



Figure 3. A node of G.

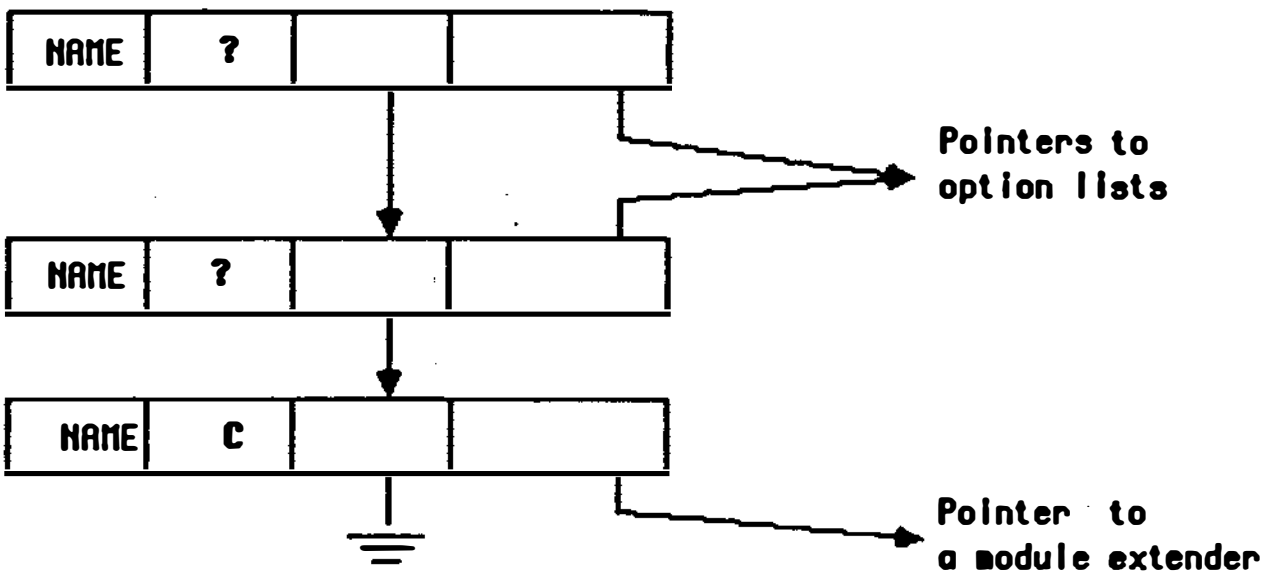



Figure 4. Option List with 3 outgoing categories

that point to an option list node have **TYPE** '?', and those that point to a module node have **TYPE** code 'c'. Each option list has a title which is the previous category selected along the access path to the option list node. (The option list is all the categories leaving the node.)

An option list is represented by a linked list connected by the **NEXT** pointers (Figure 4). The chain can have either option list nodes or module nodes in it, and it makes no difference until the node has to be displayed. More than one node may point to the start of a chain. There is no restriction on how cycles are formed.

# 4 OPERATIONS ON GRABBAG:

As mentioned before there are two types of users for **GRABBAG**: searchers and builders. Consequently, operations on **GRABBAG** have been grouped into two main modes: **LOOKUP MODULE** operations and **ADD MODULE** operations.

Searching and adding modules is done with a mouse. To make a selection, click the mouse button while pointing to an object on the screen.

## 4.1 LOOKUP MODULE OPERATIONS:

As shown in Figure 5, the first option list is displayed immediately after opening **GRABBAG** in a LOOKUP Module mode. The user chooses an access path by clicking categories one at a time. Once clicked, each category will be highlighted and cannot be clicked again. You can make as many selections as desired from each option list. Multiple selections are treated as parallel search paths (they are stacked for backtracking later on). The parallel search paths are traversed in the same order they are selected, so choose them in the order you want to search.

When the OK button is selected, the search moves to the next option list node. If more than one option was selected, the first one selected is the first option list retrieved. Selection continues from node to node until either no selection is made before the OK button is hit, or a module node is reached.

If no choice is made **GRABBAG** backs up to the previous node selected. For example, if B and C are choosen from option list A in Figure 1, B is displayed first leaving C to be displayed when the search backtracks. Figure 6 shows how node B of Figure 1 appears during the search. Looking at Figure 1 again, suppose node g is selected from the B option list. Since g is a module node the result is a module dialog as shown in Figure 7. The title is displayed at the top, and a double column of extenders is shown under the title.

When a module is to be inspected, only the attributes that exist are listed as choices, see Figure 7. There are two radio buttons labeled view and copy, and a standard button

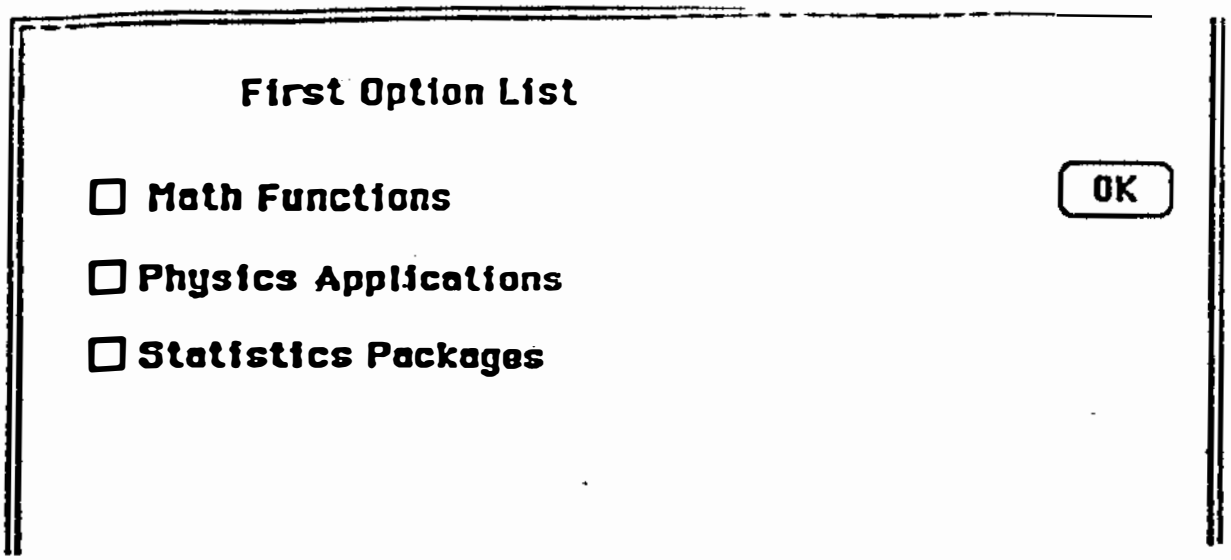



Figure 5 Option List A (refer to Figure 1)

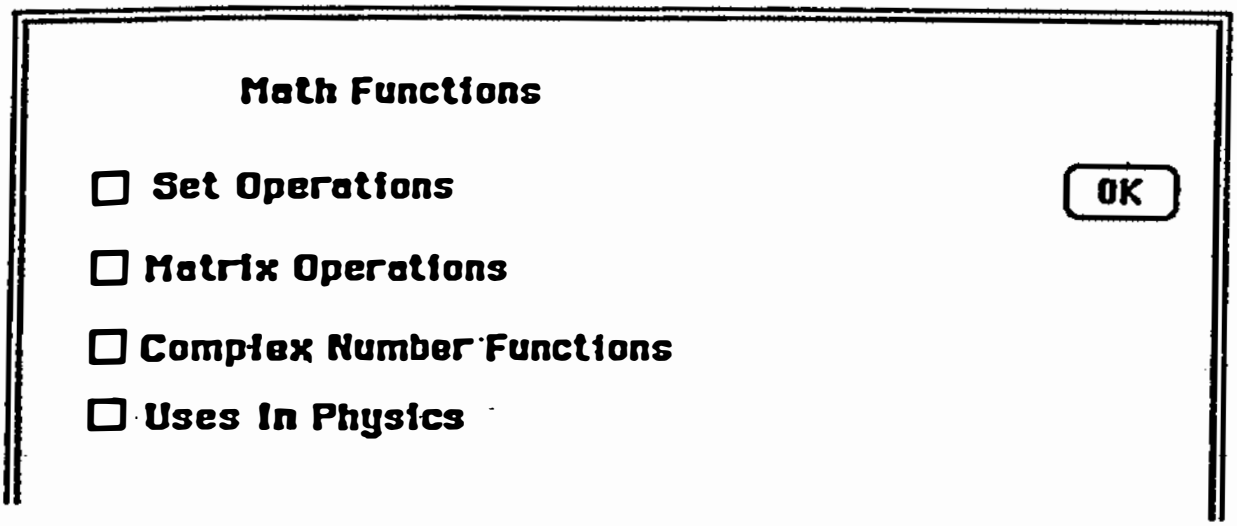



Figure 6 Option List B (refer to Figure 1)

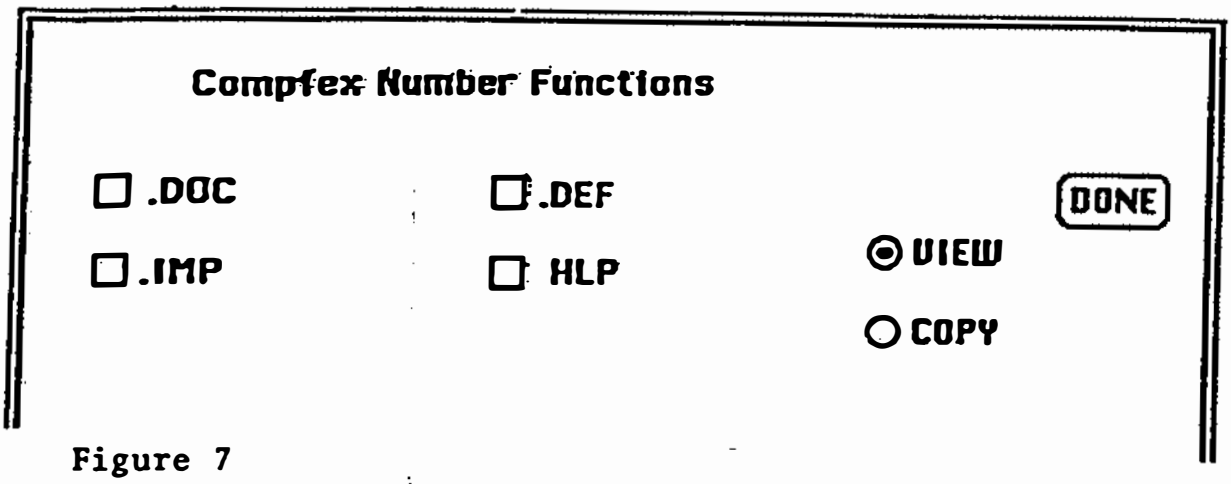



Figure 7

labeled done. The view button is preselected.

Click either view, copy, or done before clicking an attribute. When an attribute is clicked, the action indicated by the currently clicked radio button will be attempted.

When view is selected, the text from the corresponding attribute file will be displayed in a window. If copy is selected the name of the destination file to copy the database module attribute file into is requested. After the user gives a name **GRABBAG** then copies the text from the attribute file to the file you named.

When done is selected the search backtracks to option list C. If there where no backtrack nodes to search **GRABBAG** asks if you want to start again from the root.

The searcher is assumed to be able to detect cycles and take appropriate action to recover from them, i.e. you will get tired of making the same selections over and over again and try something new, or quit.

The richness of the access path interlinking and the choice of appropriate names for the categories is up to the database builder, who's actions are discussed next.

## 4.2 ADD MODULE OPERATIONS:

A builder has more flexibility and greater control than a searcher. A builder can either create a new database of reusable modules, or add modules to an existing database.

When opening a **GRABBAG** database to ADD a module an access code will be required as shown in Figure 8. After the user enters the access code, the dialog shown in Figure 9 is displayed. If the create option is clicked the dialog for creating a new database is activated.

### 4.2.1 NEW DATABASE:

A new database always starts from a system supplied root node. **GRABBAG** requests a new category corresponding to the arc connecting the root to the new node (Figure 10). **GRABBAG** then asks if the new node is a module node or an option list node (Figure 11). If you request an option list node, new categories are repeatedly requested, (Figure 10), until you request a module node. In this way, a new network of option lists is constructed, depth-first.

When a category is to have a module attached, the system creates a new module and then displays the list of all possible attributes, the final category as a title, and a done button (figure 12). Clicking an attribute will cause the attribute extender to be highlited, and the system to ask for the name of the source file to be put into the database. The user

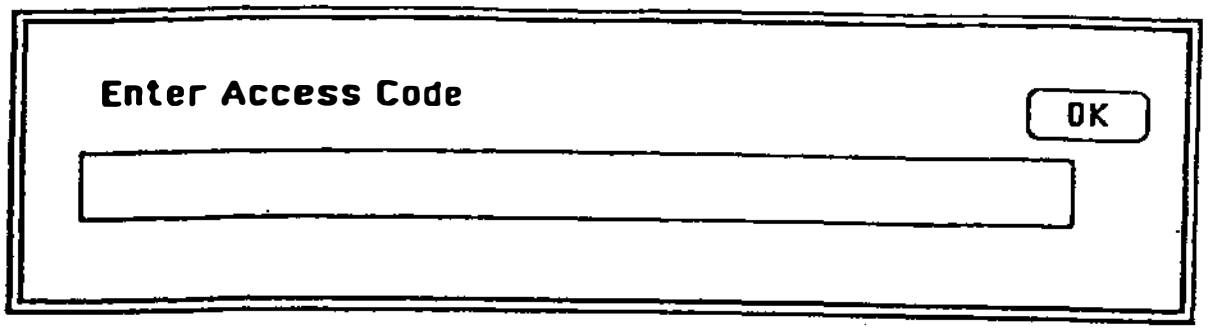



Figure 8

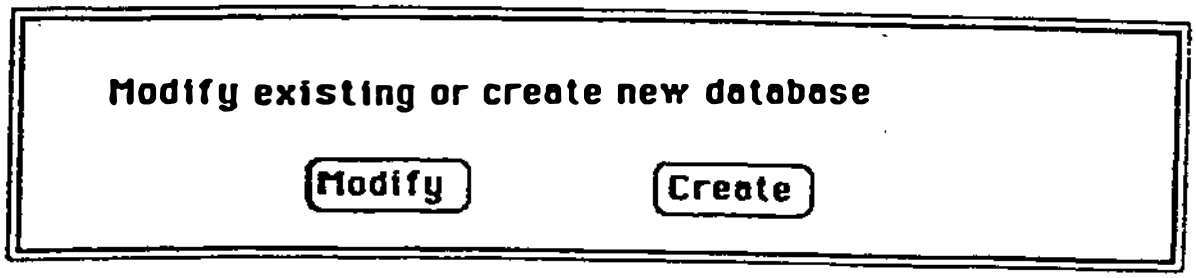



Figure 9

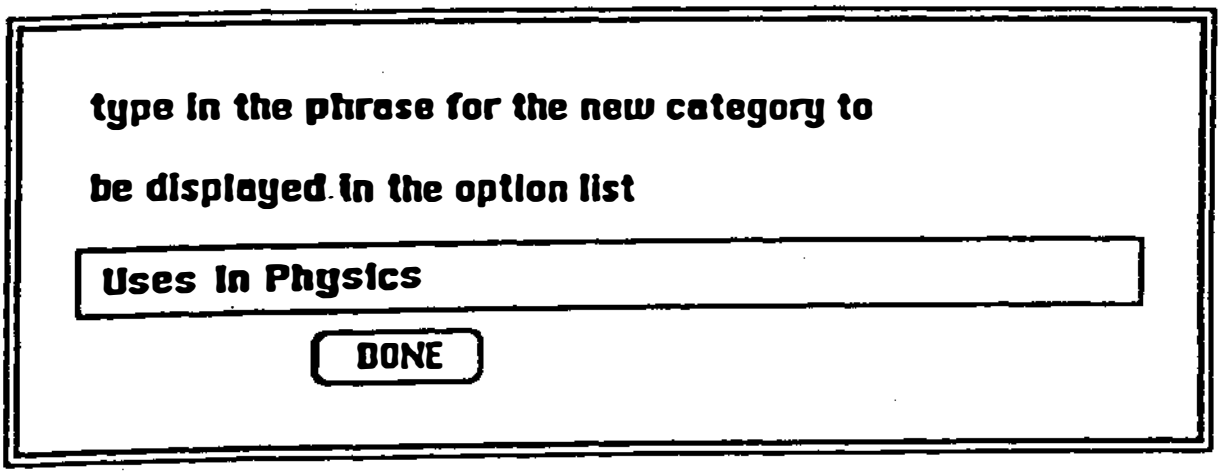



Figure 10

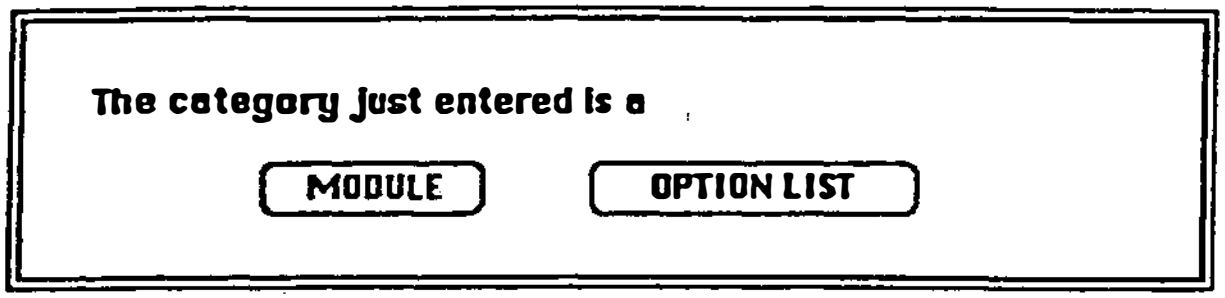



Figure 11

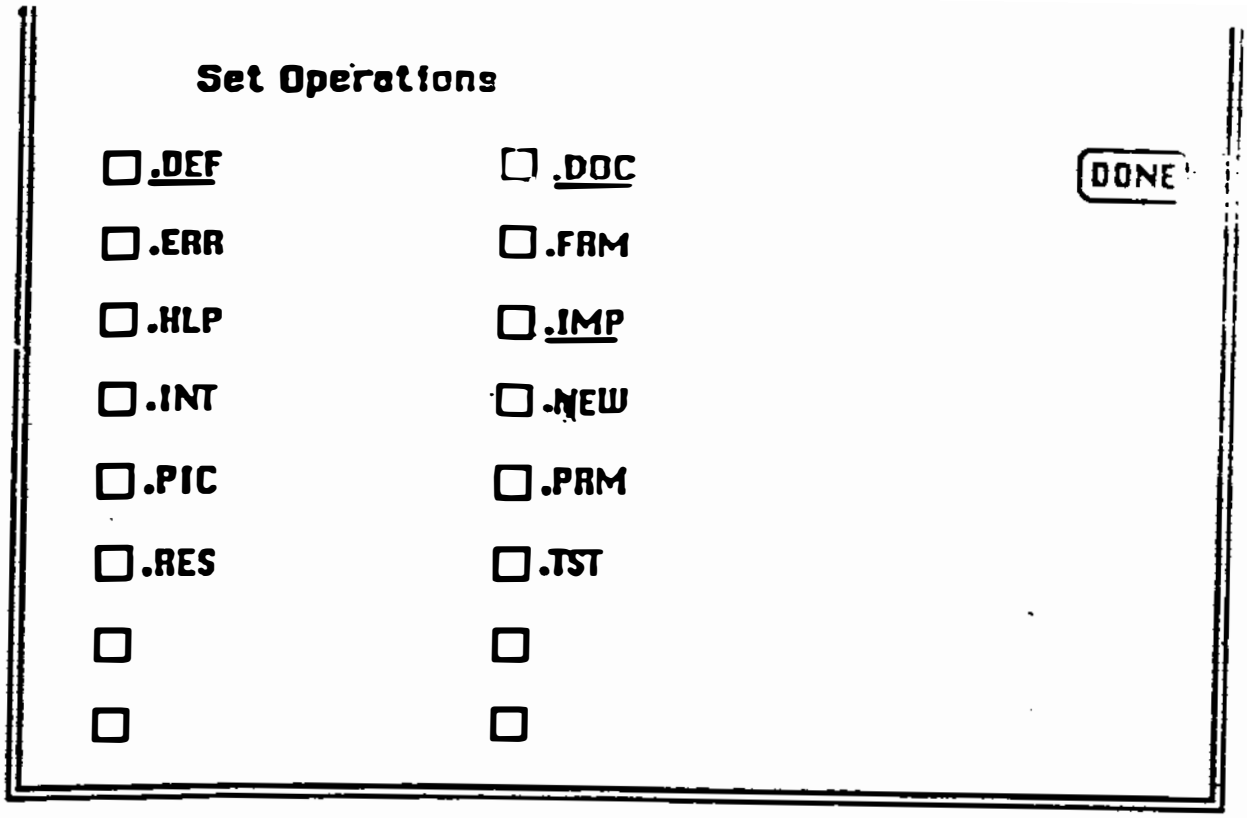



Figure 12

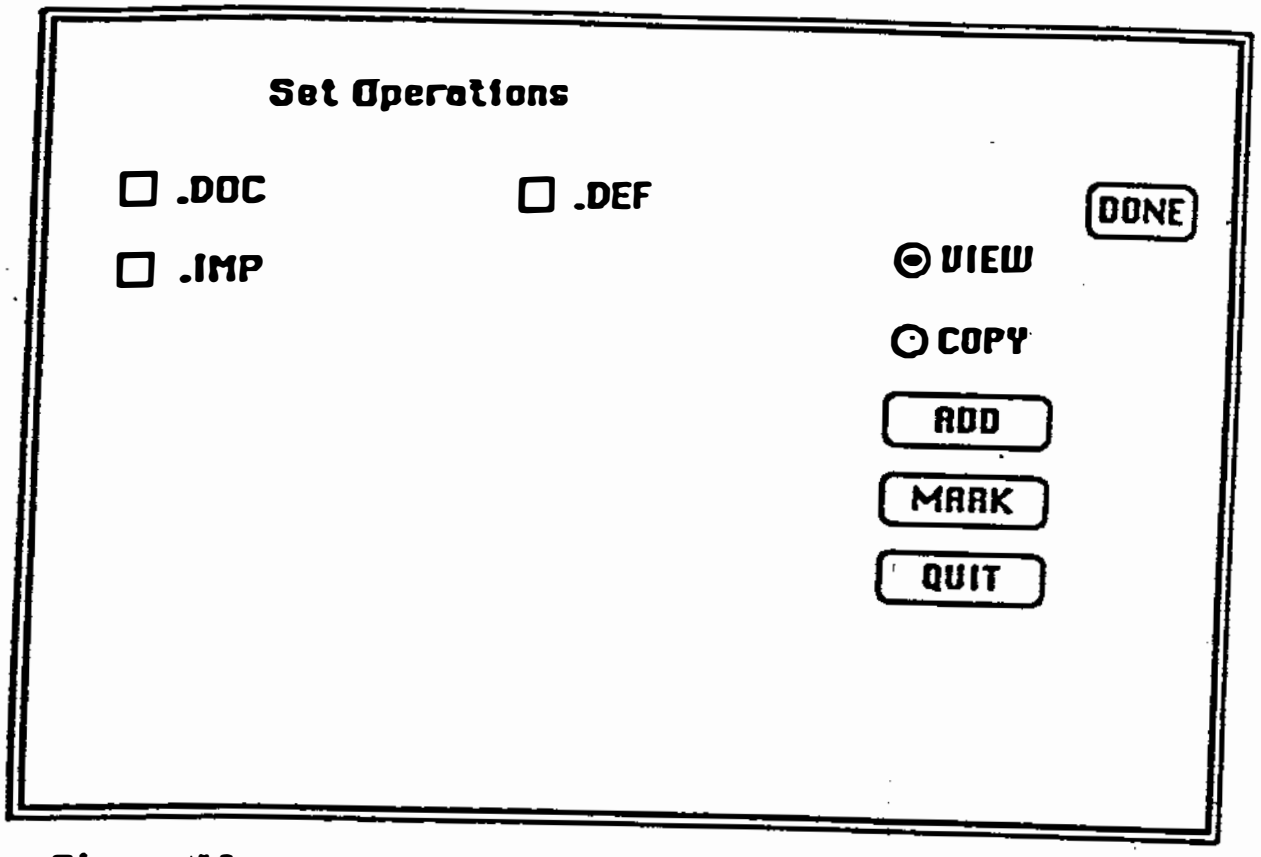



Figure 13

then copies the source file to the attribute file. GRABBAG then displays the already added attribute files, Figure 13. To add another attribute click the ADD button to get Figure 12 displayed again - select a new attribute and continue as before.

When you finish adding attributes click the DONE button. The search backtracks to the previous selected option list. To bypass the backtracking click the Quit button to return to the root node. The view and copy buttons work the same as in LOOKUP module.

### 4.2.2 EXISTING DATABASE:

When an existing database is entered by clicking the Modify option in Figure 9, the first option list node is displayed. The display has a column of actions down the right side of the screen (Figure 14). The buttons, and what they do are listed below:

#### OK

Clicking OK backtracks to the previously selected option list node. If you did not select multiple categories from previous option list nodes, Ok reverts to the root of the network.

#### CANCEL

Cancel reverts to the root node and bypasses all backtracking. Use this as a quick way to jump back to the root of the network and start over again.

#### QUIT

The Quit button allows you to bypass a stack of categories still to be displayed and quit the database. Builders may continue the session by opening **GRABBAG** again.

#### SELECT

Select is used to build one or more access paths to a module or an option list node before doing any operations on it.

#### MARK

The Mark button controls how the system can be interconnected. The mark can be thought of as a single token that always exists in the system. After creation of a new database, the root (the only node yet existing) is marked. A marked node has the mark button highlited when it is displayed.

To create an interconnection, first use select to go to the node you want to be connect to, either an option list or a module, and then mark that node. After marking the target node, either backtrack or use select to go to the node to connect from and add a category

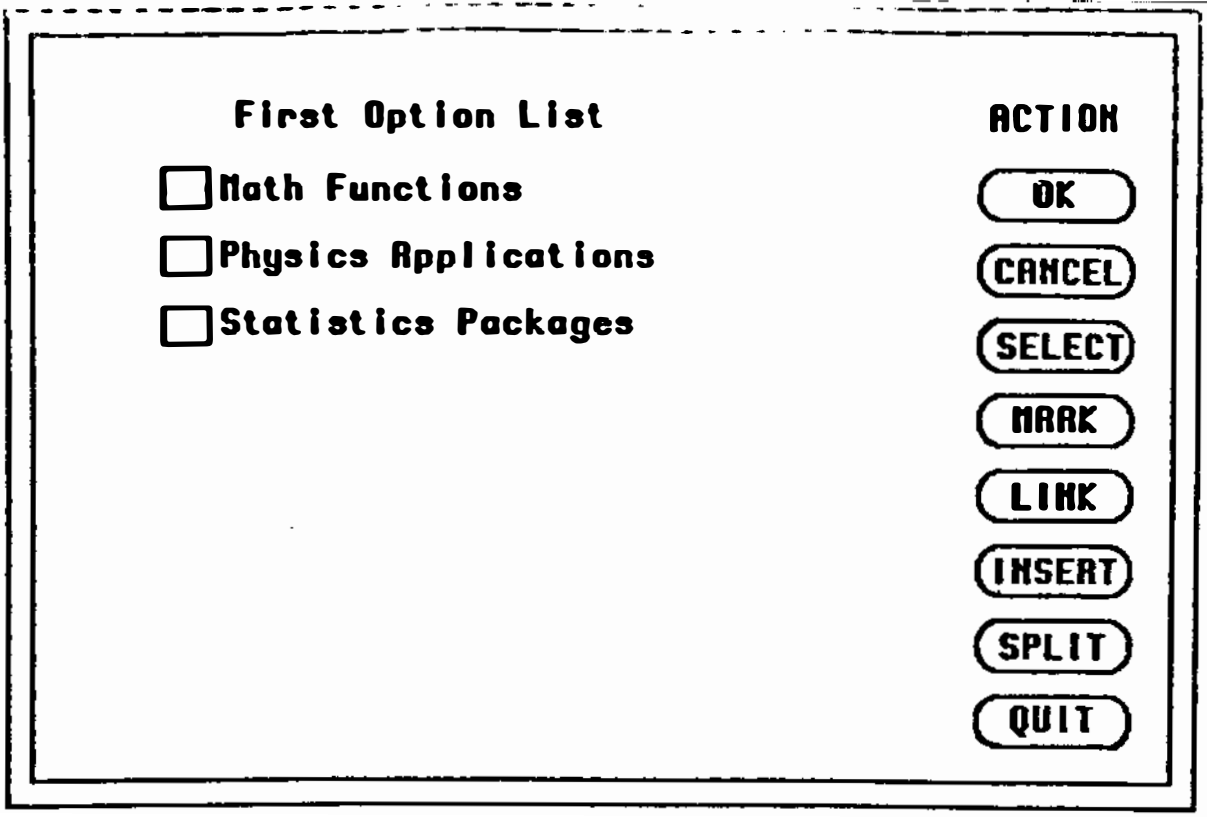



Figure 14

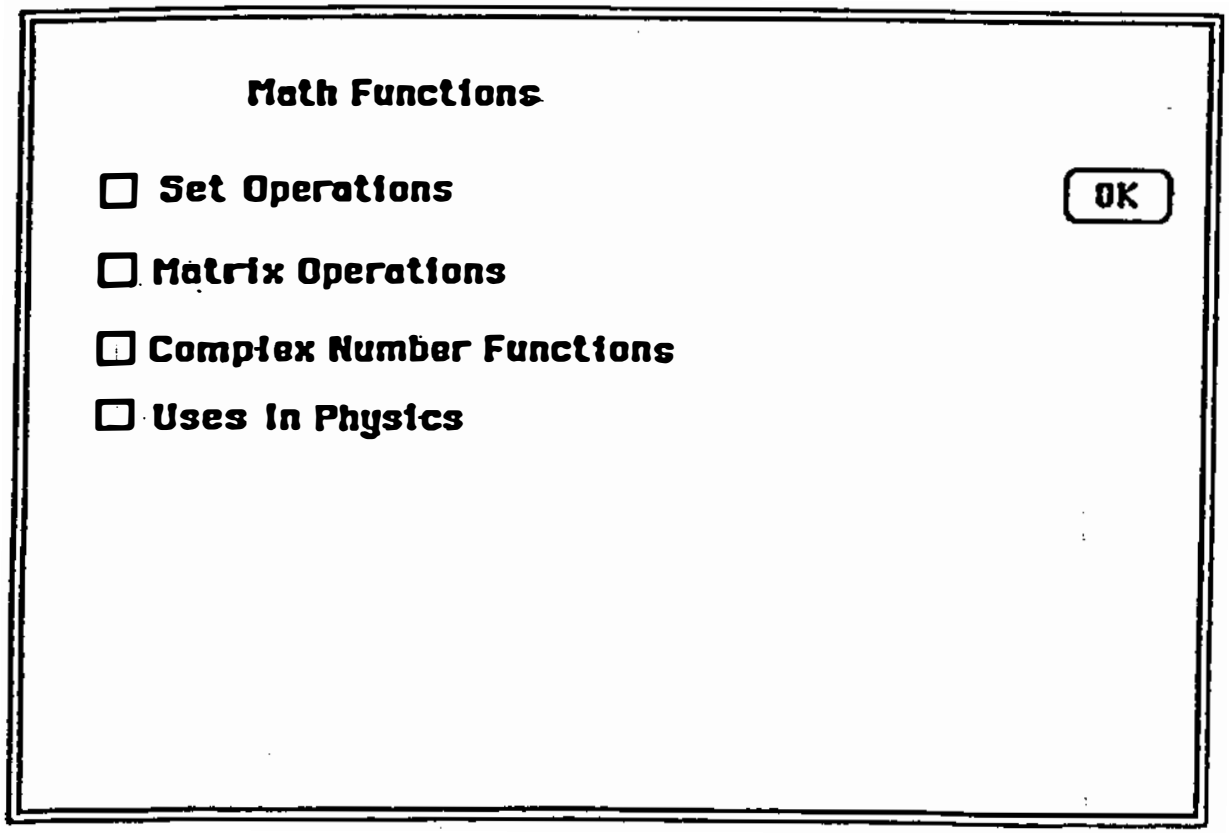



Figure 15

from there to the marked node. Use the link button to establish the link to the marked node.

## LINK

The Link button is used only when you want to connect the current node to the marked node. First you must mark a node as described above using the Mark button. The connection can be from an option list node to either another option list node or a module node.

Suppose category 12 in Figure 1 was not there and you want to link node B to node C. First select Physics Applications (C) followed by selecting Math Function (B), see Figure 5. Mark node C and click OK to backtrack to node B. Click the link button and **GRABBAG** will ask for the text of the new category, see Figure 10. Enter text into the box (Uses in Physics) and click the Done button, to get Figure 15.

## SPLIT

Split is used to take categories from the current option list and put them in a new option list that is reached through the currect option list. The new option list is hierarchically "below" the current option list in the network, Figure 16.

After clicking the split button, the system displays the current option list and waits for you to make the first category selection. Select the first category to be split and enter the name of the new option list that will accommodate the split categories. Continue selecting other categories to be moved. After you have removed all the categories that you wish to split, the system will return you to the original option list with the new entered option list name as the last category of the list that remains, Figure 16.

## INSERT

The Insert button inserts new categories into an existing option list node, or new attributes to an existing module. A new category is added to the option list immediately following the one you select. Insert groups similar items together even though they may not have been entered at the same time. For example, if you click the insert button in Figure 14, Figure 17 will be displayed. Clicking any category brings Figure 10 to the screen (of course with the field for the sentence "uses in physics" empty). The user then types a name. Consequently Figure 11 is displayed, and after this the dialog continues as described before for entering a new node.

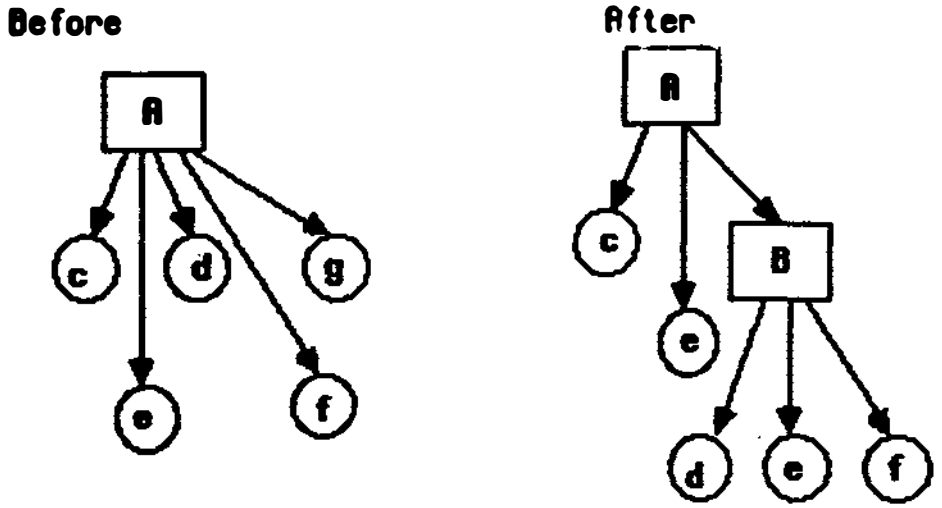
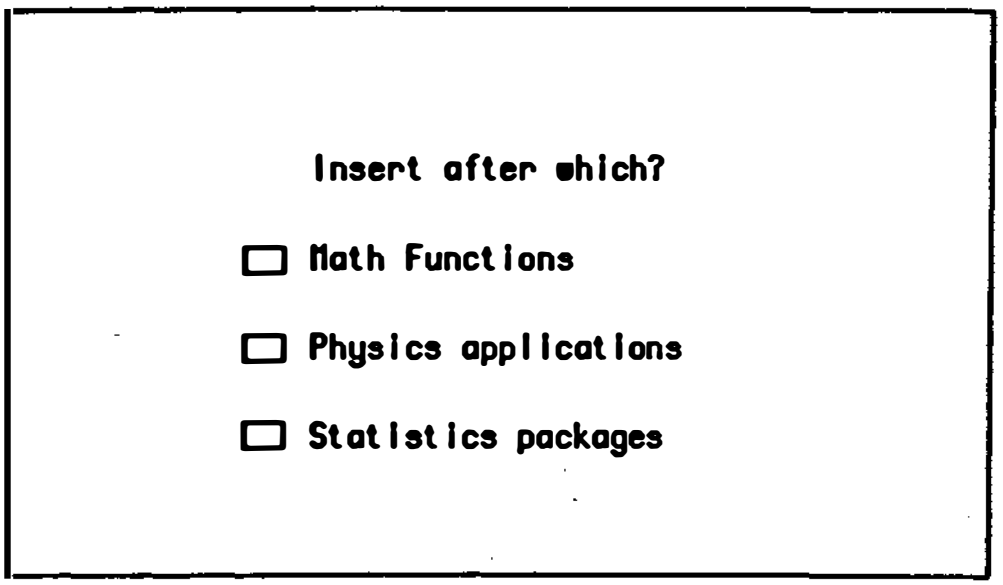



Figure 16
Split selecting d, f, and g from option list.

---



# 5 CONCLUSION

The idea of reusing software components has started to take root in many software development groups. It promises to increase productivity by one or two orders of magnitude. However, we must build libraries of standard parts which can be easily located and modified. This is currently being done within our group. We have composed libraries in Pascal and C for handling user interface objects, data structures, file I/O; and miscellaneous operations.

In addition, the original **GRABBAG** system, which was written in Modula-2, is currently being re-designed and written in Pascal. The new design incorporates improvements which we learned were needed after considerable use of the system. We are implementing the new system in Pascal in order to make lookup operations work as a desk accessory. This allows a programmer to use the lookup operations from within an editor or other application.

Future developments include an automatic generalizer and refiner tool which takes modules from the programmers' database, generalizes them, and then specializes them to serve a new purpose. This technique is very useful for restructing source code and reuse it. For example, a valuable module in Modula-2 might be restructured to work on a different data type, or perhaps re-written entirely in another programming language.

The term aliasing which is usually a nuisance (in programming languages) turned out to be extremely beneficial because it represents the interconnection between modules. However, we probably should have chosen another term instead of aliasing.

The rich set of module attributes contributes to making decisions during module design, implementation, and modification visible to users. The interface specifications stored in **GRABBAG** can be used during design of a new program. The implementation specifications can be retrieved, modified, and then used in a new program, also. Thus **GRABBAG** provides a repository for reusable code, which can improve programmer productivity manyfold over conventional manual methods.

**GRABBAG** has been successful because it performs a valuable service to software developers, it incorporates an extremely easy-to-use interface, and it serves as a focal point for programmers. Developers are loath to search through manuals to find a reusable module, but when the modules are on-line, they quickly change their habits to conform to the reusable software life cycle model.

# 6 REFERENCES


1. M. Shaw, "Abstraction Techniques in Modern Programming Languages.", IEEE Software, Vol. 1, No. 4, October 1984.

2. G. P. Brown, et al, "Program Visualization: Graphical Support for Software Development." IEEE Computer, Vol. 18, No. 8, August 1985.

3. D. R. Smith, et al, "Research on Knowledge-based Software Environment at Kestrol Institute." IEEE Software Engineering, Vol. SE-11, NO. 11, November 1985.

4. T. J. Biggerstaff and A. J. Perdis, (editors), "Special Issue on Software Reusability." IEEE Software Engineering, Vol. SE-10, No. 5, September 1984.

5. R. Kowalski, "Software Engineering and Knowledge-based Systems in New Generation Computing." FGCS, North-Holland, Vol. 1, No. 1, July 1984.

6. M. R. Barabacci, et al, "The Software Engineering Institute: Bridging Practice and Potential." IEEE Software, Vol. 2, No. 6, November 1985.

7. W. Myers, "MCC: Planning the Revolution in Software." IEEE Software, Vol. 2, No. 6, November 1985.

8. T. C. Jones, "Reusability in Programming: A Survey of the State of the Art." IEEE Software Engineering, Vol. SE-10, No. 5, September 1984.

9. T. A. Standish, "An Essay on Software Reuse." IEEE Software Engineering, Vol. SE-10, No. 5, September 1984.

10. E. Horowitz and J. B. Munson, "An Expansive View of Reusable Software." IEEE Software Engineering, Vol. SE-10, No. 5, September 1984.

11. J. M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components." IEEE Software Engineering, Vol. 10, No. 5, September 1984.

12. B. W. Kernigham, "The UNIX System and Software Resuability." IEEE Software Engineering, Vol. SE-10, No. 5, September 1984.

13. The XENIX Development System, Programmer's guide for the Apple Lisa, The Santa Cruz Operation, Inc., 500 Chestnut St./ PO Box 1900/ Santa Cruz, CA/1984.

14. I. P. Goldstein and O. G. Babrow, "A Layered Approach to Software Design." Interactive Programming Enviroments, McGraw-Hill Book Company, 1984.

15. A. Motro, "Browsing in a Loosely Structured Database." SIGMOD '84, Vol. 18, No. 2, Proceedings of annual meeting, Boston, MA, June 18-21, 1984.

16. J. Whiteside, et al, "User performance with Command Menu, and Iconic Interfaces." ACM SIGHI, CHI '85 proceedings, April 14-18, 1985.

# Session 6

## PANEL: INTEGRATED VS. INDEPENDENT SOFTWARE EVALUATION

*Chuck Asberry, Tektronix; Rich Martin, Intel Scientific Computers; David A. Rodgers, Boeing Commercial Airplane Co.; David Gelperin, Software Quality Engineering; and LeRoy Nollette, Panel Chairman*

There is a perceived relationship between the organization of the development group and the quality of the resulting products they produce. Some contractors, notably the Department of Defense, require an independent software evaluation team to "check" the output of the design organization.

Arguments in favor of this are that the development team would not be able to find some kinds of errors in their own work, and that they may be more inclined to yield to deadline pressures at the expense of quality.

Others feel that the quality of the output is actually the responsibility of the development team and that rather than having an external group finding errors, the evaluation engineering function should reside within the development team.

Both forms of evaluation exist today and both yield relatively high quality software. Clearly there is not enough evidence to suggest that one form is superior to the other.

Certainly, abuse of any organizational structure might allow for the generation of errors that would go undetected. At the same time, the entire engineering community generally strives to always produce quality products. Regardless of the organization they work in, they will go to extra effort to make sure their products meet some level of quality.

This panel discussion provides a forum for the conference attendees to participate in a dialogue with professionals who can share the real-world experiences they have had in their respective structures.