SEVENTH ANNUAL PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

September 10-12, 1989

Red Lion/Lloyd Center
Portland, Oregon

TABLE OF CONTENTS

Chairman's Message

Chuck Martiny

As the software development profession matures, improvements are taking place at an ever-increasing rate. For several years, we have focused on improving our development tools but recently, we have seen more attention devoted toward improving the development **process** as well. A primary objective of the Pacific Northwest Software Quality Conference is to share information that will increase our professional growth and in turn, our contribution to the profession and our companies or universities. Each year, we have made changes that improve the conference. This year we have added tutorials and are recognizing outstanding contribution to our profession with the Software Excellence Award.

This is the seventh year of the conference and the papers are the best ever submitted. We have divided the presentations into one management track and two engineering tracks. Our keynote speaker, Capers Jones, is recognized as one of the world's leading authorities in software development and his message should benefit everyone at the conference. Combining the keynote, excellent papers, tutorials, the paper for the Software Excellence Award, the workshops, and the commercial exhibits, we are confident that this will be the best conference we have ever offered.

Our intent is for everyone to learn something that will make a lasting contribution in their professional careers.

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Chuck Martiny – President and Chair
Tektronix, Inc.

Dick Hamlet – Technical Program
Portland State University

Sue Bartlett – Secretary; Workshops
Mentor Graphics Corp.

Steve Shellans – Treasurer; Keynote
Tektronix, Inc.

Mark Johnson – Exhibits
Mentor Graphics Corp.

Paul Blattner – Board Member
Quality Software Engineering

Doug Saxton – Board Member
Portland General Electric

CONFERENCE PLANNING COMMITTEE

Sandy Baldrige
Sequent Computer Systems

Kristina Berg,
Mentor Graphics Corp.

Paul Blattner
Quality Software Engineering

Greg Boone
CASE Research Corp.

John Burley
Mentor Graphics Corp.

Alan Crouch
Tektronix, Inc.

Margie Davis
ADP Dealer Services

Greg Deffenbaugh
Boeing Computer Systems

Dave Dickmann
Hewlett Packard

Bill Edmark
Intel Corp.

Sara Edmiston
Intel Corp.

Rick Frankel
Seattle University

Cynthia Gens
Portland State University

Micheal Green
Portland State University

Dick Hamlet
Portland State University

Elicia Harrell
Intel Corp.

Warren Harrison
Portland State University

Bryan Hilterbrand
Tektronix, Inc.

Mark Johnson
Mentor Graphics Corp.

Bill Junk
University of Idaho

Richard D. Lane
Tektronix, Inc.

Jim Larson
Tektronix, Inc.

Cham Lee
Tektronix, Inc.

Ben Manny
Intel Corp.

Kent Mehrer
Mentor Graphics Corp.

Michael Meyer

Dale Mosby
Sequent Computer System, Inc.

Shannon Nelson
Intel Corp.

LeRoy Nollette
Metro

Ken Oar
Hewlett Packard

Don Payne
Ben Franklin

Misty Pesek
Quantitative Technologies Corp.

Jeff Ruby
Tektronix, Inc.

Dennis Schnabel
Mentor Graphics Corp.

Len Shapiro
Portland State University

Rick Socia
Spacelabs

Eugene Spafford
Purdue University

Sue Strater
Mentor Graphics Corp.

Bill Sunderrneier
CADRE Technologies, Inc.

Craig Thomas
Mentor Graphics Corp.

Lee Thomas
Quantitative Technologies Corp.

George Tice
Mentor Graphics Corp.

Donna Warner
Intel Corp.

Nancy Winston,
Intel Corp.

Barbara Zanzig
Tektronix, Inc.

PROFESSIONAL SUPPORT

Lawrence & Craig, Inc.
Conference Management

Liz Kingslein
Graphic Design

Decorators West, Inc.
Decorator for Exhibits

Authors

Dennis de Champeaux
Hewlett-Packard
HP Labs, 3U
PO Box 10490
Palo Alto, CA 94303-0971

Kenneth Dickey
Tektronix, Inc.
PO Box 4600, MS 92-101
Beaverton, OR 97076

Roger J. Dziegiel, Jr.
Rome Air Development Center
Software Engineering Branch (COEE)
Griffiss AFB, NY 13441-5700

Richard E. Fairley
Marilyn Bush
SITE, Rm.203
George Mason University
4400 University Drive
Fairfax, Virginia 22030

Leona F. Fass
P.O. Box 2914
Carmel, CA 93921

Christopher Fox
Room 2H-536
AT&T Bell Laboratories
Holmdel, NJ 07733

Modenna Haney
Regina Palmer
Phil Daley
Martin Marietta Astronautics Group
Space Systems Company
P.O. Box 179 M.S. 4372
Denver, CO 80201

C. Frederick Hart
Georgia Institute of Technology
PO Box 37174
Atlanta, GA 30332

Daniel Hoffman
University of Victoria
Dept. of Computer Science
P.O. Box 1700
Victoria, B.C., Canada V8W 2Y2

Herb Krasner
Mike Pore
Lockheed Software Technology Center
0/96-10 B/30E
2100 E. St. Elmo Road
Austin, TX 78744

Patrick H. Loy
Loy Consulting, Inc.
3553 Chesterfield Avenue
Baltimore, MD 21213

Peter M. Martin
Richard Dizmang
Jeff Crawford
Eric Anderson
Apple Computer Inc.
20525 Mariani Ave
Cupertino, CA 95014

Dag Mellgren
CAP SESA Innovation
Centre de Recherche de Grenoble
33, Chemin du Vieux Chene, ZIRST
F-38240 Meylan (France)

Walter Olthoff
Dennis de Champeaux
Hewlett Packard Labs
1501 Page Mill Road
Palo Alto, CA 94304

Northon Rodrigues
William Tracy, Jr.
Gregory Miller
Prime Computer, Inc.
Computervision Electronics
Development Center
14952 NW Greenbrier Parkway
Beaverton, OR 97006-5733

Robert E. Shelton
CASEware, inc.
P.O. Box 8669
Portland, OR 97201

George Stevens
ADVANCE Computer Engineering, Inc.
P.O. Box 10736
Portland, OR 97210

Jason Su
Corvallis Workstation Operation
Hewlett-Packard Company
1000 NE Circle Boulevard
Corvallis, OR 97330

K. C. Tai
Department of Computer Science
North Carolina State University
Box 8206
Raleigh, NC 27695-8206

Ross Taylor
Tektronix, Inc.
P.O. Box 1000, M/S 63-356
Wilsonville, OR 97070-1000

Mladen A. Vouk
Robert E. Coyle
North Carolina State University
Dept. of Computer Science, kBox 8206
Raleigh, N.C. 27695-8206

Elaine J. Weyuker
Dept. of Computer Science
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, New York 10012

Tze-Jie Yu
AT&T Bell Laboratories
Naperville, IL 60566

Richard E. Zultner
Zultner & Company
12 Wallingford Drive
Princeton, NJ 08540-6428

Exhibitors

Nina Wachsman
APPLIED BUSINESS TECHNOLOGY
149 New Montgomery St. #504
San Francisco, CA 94105
415/882-9974

Bill Sundermeier
CADRE TECHNOLOGIES
19545 NW von Newmann Drive
Beaverton, OR 97006
503/690-1300

Cheryl Holden
INDEX TECHNOLOGY
One Main Street
Cambridge, MA 02142
617/494-8200

Hwe-Chu Tu
INSTITUTE FOR ZERO DEFECT SW
200 Runnymede Parkway
New Providence, NJ 07974
201/604-8701

Vicky Carman
INTEL CORPORATION
5200 NE Elam Young Parkway
Hillsboro OR 97124
503/696-2484

Diana Felde
KNOWLEDGEWARE, INC.
3340 Peachtree Road NE, #2900
Atlanta, GA 30326
404/231-3510

Lisa Powell
POLYTRON CORPORATION
1700 NW 176th Place
Beaverton, OR 97006
503/645-1150

Shawn Wall
POWELL'S TECHNICAL BOOKS
32 NW 11th
Portland, OR 97209
503/228-3906

Gail Anderson
SEMAPHORE TRAINING
800 Turnpike Street, #300
North Andover, MA 01845
503/794-3366

Edward Miller
SOFTWARE RESEARCH, INC.
625 Third Street
San Francisco, CA 94107-1997
415/957-1441

David Gelperin
SOFTWARE QUALITY ENGINEERING
4825 Olson Highway, #240
Minneapolis, MN 55422
612/541-1431

Chris Francis
VERILOG USA
Beauregard Square
6303 Little River Turnpike
Alexandria, VA 22312
703/354-0371

SOFTWARE QUALITY AND INDUSTRY LEADERSHIP

ABSTRACT

U.S. industry has been losing global market shares in high technology products due to not understanding that quality rather than least cost production is the dominant factor leading to success.

Computing and software have become major factors in both the operations of modern enterprises and in the products which many enterprises market. Their importance is such that it can be asserted that the enterprises that master computing and software will be the dominant enterprises of the 21st century, while those that fall behind may not survive. The key to mastering computing and software is to strive for quality, since the technologies that control quality also tend to optimize productivity, schedules, and user satisfaction.

Five distinct "waves" of software have emerged since the industry began, with each wave having its own characteristic profile of significance and associated technologies. Leading edge enterprises are taking 12 specific steps to bring software under executive control, starting with enterprise wide surveys of both future needs and current capabilities. Also included are the establishment of enterprise wide measurement programs and the setting of numeric targets for software quality and productivity.

Capers Jones, Chairman
Software Productivity Research
Cambridge, Massachusetts

Capers Jones has dedicated the last 20 years of his life to the science of programming productivity. His career-long goal has been the achievement of optimal productivity and quality through the use of leading-edge tools and technologies. Mr. Jones is Chairman and co-founder of Software Productivity Research, Inc., in Cambridge, Mass., and has developed a set of expert systems for predicting software schedules, productivity, quality and reliability. Mr. Jones is an international private consultant, speaker, seminar leader and author. One of his publications for the IEEE, "Programming Productivity: Issues for the Eighties", remains one of their ten best sellers.

Kaizen Software Tools

Christopher Fox

AT&T Bell Laboratories
Lincroft, New Jersey 07738

Abstract

Kaizen is the Japanese quality philosophy of continuous improvement of all aspects of work by everyone in an organization. The Kaizen approach to quality and to process and product improvement is very different from the dominant approach to software quality assurance in the United States. This difference in approach has consequences for all aspects of software engineering; in this paper we focus on some of the consequences of these differences for the design and use of software tools to aid software engineers in improving their products and processes (Kaizen tools).

Biographical Sketch

Christopher Fox is a Member of the Technical Staff at AT&T Bell Laboratories in Lincroft, New Jersey, where he is currently working on multimedia workstations and information systems. His interest in software engineering and software quality began when he was a member of the Bell Laboratories Quality Technology Center, where he did research and development of software quality and productivity tools and methods. His current research interests in software engineering include hypermedia systems and CASE tools for managing development project work products.

Dr. Fox has B. A. and M. A. degrees in Philosophy from Michigan State University, and an M. S. in Computer Science and a Ph. D. in Information Science from Syracuse University.

Kaizen Software Tools

Christopher Fox

AT&T Bell Laboratories
Lincroft, New Jersey 07738

1. Introduction

Kaizen is the Japanese quality philosophy of continuous improvement of all aspects of work by everyone in an organization[1]. The Kaizen approach to quality and to process and product improvement is very different from the dominant approach to software quality assurance in the United States[2]. This difference in approach has consequences for all aspects of software engineering; in this paper we focus on some of the consequences of these difference for the design and use of software tools to aid software engineers in improving their products and processes (Kaizen tools).

The first section of the paper discusses Kaizen, contrasting it with the dominant view of quality in software engineering, and sketching the advantages of the Kaizen approach. The second section draws out some of the consequences of the Kaizen approach by considering three examples of tools for software quality improvement. The next section outlines general requirements for the design and use of Kaizen tools. The final section is our conclusion.

2. The Kaizen Approach

As noted in the introduction, *Kaizen* is continuous improvement of all aspects of work by everyone in an organization. This apparently simple formula has three parts:

- Kaizen emphasizes *continuous improvement.* This contrasts with the typical attitude of improving things in spurts to meet "quality objectives" mandated by management. Giant improvements in software quality are unlikely[3]. Worse yet, the *status quo* tends to deteriorate without continuous improvement efforts. Kaizen emphasizes steady, gradual improvement through better techniques, methods, and tools, continuous education, and critical attention to all aspects of software engineering processes.
- Kaizen makes *all aspects of work* targets for improvement. The technology available to software engineers has traditionally been regarded as the main focus of software quality improvement efforts. However, some researchers[4] (and certainly all practitioners) know that many other factors, like noise in the workplace, privacy, table space, freedom from interruption, etc., have profound effects on quality and productivity. Environmental factors may have a larger influence on quality and productivity than technical factors. Kaizen encourages focus on *all* workplace factors because all workplace factors can affect quality.
- Kaizen assigns responsibility for quality improvement to *everyone* in an organization. This contrasts sharply with the usual approach of leaving quality assurance to a quality assurance organization or a testing and validation organization. Under Kaizen, everyone is responsible for *building quality in* to the products of their work, obviating the need to *add quality on* at the end of the job. Building quality in to a product is more effective and less expensive than adding quality on to a product once it is complete; it is well known that finding and correcting errors early in the software life cycle is orders of magnitude cheaper than finding and correcting them late in the life cycle[5].

With this brief introduction, we can consider some of the consequences of adopting the Kaizen approach in software engineering. We begin with a thumbnail sketch of current software quality assurance practices. This characterization is based on published surveys[6] [7] [8], and personal observations.

1. Software quality is primarily the responsibility of a quality assurance organization and a testing organization. Improvement is not the responsibility of most project staff, who are mainly concerned with producing work products according to schedule. (A notable exception to this generalization is the use of walkthroughs and inspections, which do make improvement the responsibility of work product originators.)

2. The quality assurance organization audits project products and processes, and collects data. From these inputs it validates work products, recommends changes in development processes, and reports quality metrics to management. This role often fosters an adversarial relationship between the development and quality assurance staff. The quality assurance organization may be regarded as the "quality police" by the rest of the organization.

3. The testing organization conducts system test during the testing phase of the software life cycle. System test may be the first time that a software system undergoes careful scrutiny. Hence the testing organization may bear the brunt of an effort to add quality to a virtually finished product.

4. Quality goals are often expressed in terms of quality metrics. This tends to encourage *ad hoc* improvements until products meet their metrics goals, then improvement efforts stop.

5. Efforts to improve quality and productivity tend to focus on technical innovations, such as the latest language, debugger, editor, windowing system, etc. Even so, there is often hesitation to adopt new innovations without overwhelming proof that they will improve quality metrics.

6. Software quality assurance has a bad name in many software development shops. Collecting data for the quality assurance organization is an onerous (and occasionally dishonest) task. Feedback from the quality assurance organization is often ignored. System test is a long and painful process that finds many faults, but still often fails to keep enough faults out of the final product.

How would a Kaizen approach change this picture? We consider each point in turn:

1. Dispersal of responsibility from the quality assurance and testing organizations to everyone changes everyones perceptions about quality, and their roles in the organization. Attention is concentrated less on meeting deadlines at any cost and more on working smarter to produce better products.

2. The quality assurance organization's role changes from quality policeman to "quality facilitator," helping everyone improve their work.

3. The testing organization is no longer the major bulwark in an effort to add quality to a finished product, but a final check of the quality already built into the product.

4. Quality improvement is no longer a fire-fighting activity to achieve required quality metric target values, but a sustained effort to monitor work practices and make changes for the better whenever and wherever anyone sees an opportunity to do so.

5. Efforts to improve quality are broadened to include important factors too often ignored, with the consequence of substantial quality and productivity gains from small changes in the workplace.

6. If the Kaizen approach works as well in software engineering as it has in manufacturing and service industries, software quality assurance will achieve its goals more cheaply, effectively, and thoroughly.

More detailed discussion of the Kaizen approach, its application in software engineering, and its consequences for software quality assurance, are beyond the scope of this paper. The remainder of the paper discusses the consequences of the Kaizen approach for the design and use of tools specifically for software engineering process improvement.

3. Kaizen Tools: Three Examples

Under Kaizen, software engineers are encouraged to monitor their own and each others processes, and to come up with ways to improve them. Software tools can help with this self-monitoring and improvement effort by measuring the software development process. A *Kaizen tool* is a software tool that monitors the software engineering process as part of a continuous effort to improve it. Many software tools already exist, and (as we will see below), some of these can be used as Kaizen software tools. Unfortunately, many development shops do not use many software tools at all beyond editors, compilers, and debuggers. A first step in process improvement in such environments is to learn about and obtain a suite of software tools, and incorporate them into the

development process. Once this is done, a continuous improvement effort using Kaizen software tools can begin.

In this section we discuss three examples of applying the Kaizen approach to the use of software tools to monitor and improve software engineering processes. To set the stage for this discussion, we first consider a few points about measuring processes.

3.1 Measuring Software Engineering Processes

Products and processes are measured using *metrics*[9]. Products are usually measured directly. For example, a software product can be measured directly by counting the lines of code or documentation, timing execution, and keeping track of failure rates. In contrast, processes can be measured directly or indirectly. Direct process metrics include measures of developer activity, such as the number and frequency of compilations, measures of developer characteristics, such as years of experience, and measures of the process environment, such as the noise level in the workplace. Indirect process metrics are obtained by measuring products during the development process. For example, measures of fault density, comment density, and complexity made during development can be used to study the development process itself.

Kaizen tools measure processes. They may do so directly or indirectly. The Kaizen tools we discuss in this section all produce indirect process metrics, but that is only because it is usually easier to write software tools to measure products (and hence measure processes indirectly) than to measure processes directly. Furthermore, the tools we discuss in this section are all UNIX®* tools, but that is only an accident of our development environment. Kaizen software tools can be developed for, and used in, any software engineering environment.

3.2 Using `Lint` as a Kaizen Tool

`Lint`[10] is a C program checker that finds likely sources of bugs, inefficiencies, and non-portability. Although `lint` finds errors that the C compiler would also find, its primary value is as a means of finding errors that the C compiler would not find. To illustrate, Figure 1 shows a portion of the `lint` output for an example C program:

* UNIX is a registered trademark of AT&T.

```
example.c
=============
(164)  too many characters in character constant
(1032)  warning: u unused in function rpack
(1223)  warning: fp redefinition hides earlier one
(1435)  warning: ret unused in function get_x_char
(1762)  warning: m set but not used in function chk10
(164)  static variable delete unused
(1)  static variable Csccs unused

=============
name defined but never used
    clkint          example.c(949)
    Csccs           example.c(1)
    delete          example.c(164)
value type declared inconsistently
    exit     llib-lc(46) :: example.c(1512)
    strcat          llib-lc(425) :: example.c(1572)
function argument ( number ) used inconsistently
    write( arg 3 )       llib-lc(173) :: example.c(990)
    write( arg 3 )       llib-lc(173) :: example.c(1012)
    signal( arg 2 )      llib-lc(128) :: example.c(1518)
function called with variable number of arguments
    tranlog         example.c(1615) :: example.c(304)
function returns value which is always ignored
    ungetc       sprintf      strcpy
    strcat       pclose       ioctl
function returns value which is sometimes ignored
    lget_x_char         fgets         sscanf
```

Figure 1: An Example of `lint` Output

Although **`lint`** does its job well, many C software developers do not use it at all. Others use it only because they are required to do so by their project methodology. For example, many project methodologies require that code pass **`lint`** without complaint before the code can be inspected, or before the code can be integrated. In such circumstances it is common practice to run code through **`lint`** just before it is inspected or turned over, then patch the code as necessary.

Using **`lint`** in this way is not using it as a Kaizen tool because, used in this way, **`lint`** is not monitoring the development process. Process improvement is hardly possible with data from **`lint`** obtained in this way; such data can only be used to measure the product as part of an effort to add quality on at the end of the development process.

When used as a Kaizen tool, **`lint`** is run frequently on code as it is written; often **`lint`** will be rerun after only a few lines are changed or added, in case even minor modifications introduce a problem. When used in this way, **`lint`** monitors the programming process, providing feedback that can be used for immediate improvement. Constant use of **`lint`** quickly shows bad programming habits that the programmer can then work to eliminate. For example, frequent use of **`lint`** might show that a programmer tends to ignore return values that should be checked, or to declare unused local variables. The programmer can then work to modify his or her behavior.

Thus, besides improving the current program, using `lint` this way can change the programmer's habits, improving the programming process.

3.3 Using Writer's Workbench as a Kaizen Tool

Writer's Workbench[11] is a collection of UNIX tools for analyzing English text. Writer's Workbench contains tools that check spelling, punctuation, grammar, capitalization, phraseology, and clarity. Writer's Workbench detects many errors that even the best human proofreaders tend to miss, and is a surprisingly effective tool for improving documents. To illustrate, Figure 2 shows portion of the Writer's Workbench output for an earlier version of this paper:

```
Possible spelling errors in ktools are:

CJF                 Novermber           lget
Csccs               Yeh                 prducts
Marciniak           ingore

beginning line 18 ktools
The Kaizen approach to quality and to process and
product improvement is *[ very]* different from the dominant
approach to software quality assurance in the United States.

beginning line 289 ktools
Although lint does its job *[ very]* well, many
software developers do not use it at all.

------------   Table of Substitutions   --------------

PHRASE                      SUBSTITUTION
all of: use "all" for " all of"
very: use "OMIT" for " very"
```

Figure 2: An Example of Writer's Workbench Output

Most people who prepare documents using the UNIX operating system do not use Writer's Workbench. Others use it from time to time, or more frequently if they are required to do so by their project methodology. As with `lint`, Writer's Workbench is often run just before a product is turned over to clean up problems at the end of the document production process.

When used in this way, Writer's Workbench is not being used as a Kaizen tool. When used as a Kaizen tool, Writer's Workbench is run frequently during document composition to catch errors as they are being made, and to sensitize writers to bad habits that they can work to eliminate. For example, frequent use of Writer's Workbench reveals tendencies to split infinitives, write long sentences, overuse the passive voice, etc. Writer's Workbench can improve the process of document preparation when it is used as a process monitoring Kaizen tool rather than a product measuring tool at the end of the development process.

3.4 Ccount — A Kaizen Tool

Ccount[12] is a tool designed from the start to be a Kaizen tool for programmers. `ccount` counts commentary and non-commentary C source lines, and computes comment to code ratios. These values are generated for individual functions and for code outside any function. Totals for the entire source file are also generated. Figure 3 shows `ccount`'s output for an example C source code file called `list.c`:

```
list.c          Wed Mar 15 15:35:50 1989

      Function            CSL    NCSL    CSL/NCSL
-----------------------------------------------------
         Create_Node       22      16        1.38
        Destroy_Node       12      10        1.20
      Is_Empty_List        10       5        2.00
        Create_List        10       5        2.00
     Append_Element        16      16        1.00
     Delete_Element        21      15        1.40
           external        33      16        2.06
              total       124      83        1.49
```

Figure 3: An Example of `ccount` Output

`ccount` is intended to be run during coding to make sure that functions and modules do not grow too large, and to make sure that comment density is high enough. It can help programmer's to overcome bad habits of writing long functions or compile modules, or of neglecting to write comments.

Most source code metrics tools like `ccount` are used as product measuring tool at the end of the development process, and `ccount` can be used in this way too. However, `ccount` is intended to be used as a Kaizen tool during the development process. Several design features of `ccount` attempt to encourage its use as a Kaizen tool:

- `ccount` is fast, usually generating its output in under a second.
- `ccount`'s output is simple, easy to understand, and formatted for readability.
- `ccount` reports data of immediate use in programming.
- `ccount` reports its data so that it is easy to pinpoint the part of the product with a problem. Specifically, `ccount` reports data on a per-function basis, allowing programmers to go directly to a function with a problem.

Thanks to these design features, `ccount` has several advantages over other tools not specifically designed for use as Kaizen tools. For example, both `lint` and Writer's Workbench generate reports that are long and difficult to read, include information of marginal value, report problems in ways sometimes difficult to relate to the specific aspects of the product and the process, and have slow response times. Nevertheless, all the tools mentioned in this section are useful and valuable as Kaizen tools.

4. Designing and Using Kaizen Tools

Kaizen tools measure processes. As discussed above, this can be done in two ways: measure the process directly, or measure the product to measure the process indirectly. Kaizen tools can measure processes in either way. Whether a Kaizen tool measures a process directly or indirectly, it will not achieve its aim if it is not used. Hence the main requirement of a Kaizen tool is that it have features that encourage its frequent use. We can

generate several specific guidelines from this general requirement:

- Kaizen tools should intrude as little as possible on the normal course of the process. If a tool is intrusive, it will probably not be used. Ideally, a Kaizen tool should run in the background, reporting its observations continually or whenever asked to do so. For example, a Kaizen code analysis tool could be part of an editor, perhaps displaying its results in a separate window updated whenever source code is changed. If a Kaizen tools must run as a separate program, it should at least be easy to run, and should run quickly.
- Kaizen tools should, if possible, be fun to use. A tool that makes an attractive presentation, with a clean report layout or attractive graphics, is more likely to be used frequently. A tool that reports metrics that change readily in response to user actions is more likely to be used because it gives immediate gratification. Using `ccount` is gratifying, for example, because adding comments to a function shows up immediately with improved commentary source line counts and comment to code ratios.
- Using a Kaizen tool should never have any negative repercussions. Obviously, Kaizen tools should never damage a work product. In addition, however, results of Kaizen tools should never be used to evaluate personnel; doing so is guaranteed to keep people from using the tool. If Kaizen tools are run frequently during development to track the process and to serve as a basis for improvement, there is little need to preserve their output, which then cannot be used for personnel evaluation.

Kaizen tools will also fail if they are not used properly. Again this constraint suggests further guidelines for the design of Kaizen tools:

- The output from Kaizen tools must be easy to understand. Simple and common metrics and graphics, or good/bad evaluations are best. Tool output should also be readable. For example, `ccount` reports simple metrics in a readable format, so there is little room for confusion or misunderstanding.
- The output from Kaizen tools should provide clear, specific guidance for improving the process. An excellent example of this feature is the list of word replacements generated by Writer's Workbench for bad phrases. For instance, Writer's Workbench suggests using "all" for "all of," "several" for "a number of," and omitting the word "very" altogether.

Tools written or modified to conform to these general requirements are more likely to succeed as Kaizen tools. Even the best suite of Kaizen tools must rely on users to apply the Kaizen philosophy in using them. This suggests that the final (crucial) ingredient in designing and using Kaizen tools is education in, and support of, the Kaizen philosophy.

5. Conclusion

Kaizen has been used successfully to improve quality and productivity in manufacturing and service industries. Many of the ills cured by the Kaizen approach in these industries are prevalent in software engineering, including failure to strive for continuous quality improvement, failure to disperse responsibility for quality throughout an organization, and a narrow focus for quality improvement efforts. The Kaizen philosophy has consequences for change in all aspects of software engineering. In this paper we have discussed some of these consequences for computer aided software engineering tools to improve process quality. Such tools, which we have called Kaizen tools, can help software engineers improve all aspects of their work in all phases of the software life cycle.

Acknowledgement: Thanks to Bill Frakes for ideas and discussion of many of the aspects of applying Kaizen in software engineering discussed in this paper. Thanks also to David Lubinsky for reviewing drafts of this paper.

REFERENCES

1. Imai, Masaaki, *Kaizen*, Random House Business Division, 1986.
2. Evans, Michael, and John Marciniak, *Software Quality Assurance and Management*, John Wiley and Sons, 1987.
3. Brooks, Fred, "No Silver Bullet," *IEEE Computer* 12 (4), April 1987, pp. 10-19.
4. DeMarco, Tom, and Timothy Lister, *Peopleware: Productive Teams and Projects*, Dorset House, 1987.
5. Boehm, Barry, *Software Engineering Economics*, Prentice-Hall, 1981.
6. Reifer, Donald J., Richard W. Knudson, and Jerry Smith, *Software Quality Survey*, American Society for Quality Control, November 20, 1987.
7. Hetzel, Bill, *The Complete Guide to Software Testing, Second Edition*, QED Information Sciences, Inc., 1988.
8. Zelkowitz, Marvin V., Raymond T. Yeh, Richard G. Hamlet, John D. Gannon, and Victor R. Basili, "Software Engineering Practices in the US and Japan," *IEEE Computer* 17 (6), June 1984, pp. 57-66.
9. Conte, S.D., H.E. Dunsmore, and V.Y. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings, 1986.
10. Johnson, Steven C., "Lint, A C Program Checker," AT&T Bell Laboratories, Murray Hill New Jersey, July 1978.
11. AT&T, *UNIX System V Documenter's Workbench Software Release 2.0: User's Guide*, AT&T, 1986.
12. Frakes, William, Christopher Fox, and Brian Nejmeh, *Software Engineering in the UNIX/C Environment*, Prentice-Hall, forthcoming.

A COMPARISON OF OBJECT-ORIENTED AND STRUCTURED DEVELOPMENT METHODS

Patrick H. Loy
Loy Consulting, Inc., and
The Johns Hopkins University
3553 Chesterfield Avenue
Baltimore, MD 21213
(301) 483-3532

ABSTRACT

The significance of "object-oriented" as a development method, and the current confusion over the term are addressed. A set of characteristics is proposed as a basis for agreement on a definition of the term. Object-oriented development is compared to the "structured techniques," and work in progress on integrating the two methods is reveiwed. Practical recommendations on assessing the importance of object-oriented development are given.

BIOGRAPHICAL SKETCH

Patrick H. Loy has his own software engineering consulting business, and serves part-time on the faculty of The Johns Hopkins University. His major research interests are in the areas of requirements analysis and design, and he has authored many papers on these topics. He has consulting experience in several software engineering areas, and conducts seminars in the U.S., Canada, and Europe. He also trains software acquisition managers for a U.S. government agency. He has refereed technical manuscripts for the Addison-Wesley Publishing Company, and for professional conferences and journals, including *IEEE Software*. Mr. Loy was formerly a network software analyst for Control Data Corporation's cybernet system. He is a member of the IEEE, ACM, and Computer Professionals for Social Responsibility. Mr. Loy holds the M.S. degree in computer science from The Johns Hopkins University, and the B.S. degree in mathematics from the University of Oregon.

1. INTRODUCTION

It is an undeniable truism that the methods and tools used during software development can have a significant impact on the quality of the final product. Over the past 15 years or so, the "structured techniques" have gained a great deal of acceptance as being methods that can assist the developer in the task of engineering quality into the system as it is being developed. These methods are based on a functional view of the system, with the system being partitioned according to its functional aspects.

Recently a new approach to system modeling has been gaining popularity, and many would say that it is the current state-of-the-art method for system development. This new approach, called "object-oriented," is supported by several programming languages, and several familiar names in the field of computer science endorse it as being the most advanced abstract modeling technique. Moreover, the religious zeal of some proponents of this approach has engendered an expectation that we may be heralding the arrival of the long-awaited "silver bullet " for software development.

Consequently, the subject of object-oriented development (OOD) has generated a great deal of interest among practitioners, with many dp professionals asking the questions, "what exactly is OOD?"; "how does it compare to the structured techniques?"; "is it a partial or complete life cycle method?"; and, "should we adopt it in our organization?"

Unfortunately, these questions have been very difficult to answer. Two areas of confusion lie at the heart of the problem. First, there is no universally agreed upon definition of what constitutes the OOD approach to system modeling. There are some OOD advocates, for example, who, with great conviction, tout Ada as being an object-oriented language, or even as the *premier* object-oriented language. There are others, however, who adamantly dispute that notion, declaring with equal conviction that Ada is decidedly not an object-oriented language.

The second area of confusion, which is closely related to the first, is on the differences between OOD and the structured methods. Some authors have suggested that there is a high degree of compatibility between at least some of the structured techniques and OOD. Others disagree with that notion, claiming that the differences in the modeling perspectives preclude any meaningful compatibility between the methods. Some advocates of OOD claim that it involves a more "natural" way to think than the functional approach. Some proponents of the structured techniques, on the other hand, insist that it is just as "natural" to think about functions as it is to think about objects.

This paper will address itself to the task of cutting through some of the fog surrounding this debate in an attempt to gain some clarity and perspective on the matter.

2. WHY THE ISSUE IS IMPORTANT

Many new software development methods have appeared over the past several years. Consequently, there can be a tendency to view object-oriented development as "just another method" for the developer's toolbox. However, this new approach, as evidenced by its rising popularity and the fact that the concept has matured to the point that it now encompasses a full life cycle approach, represents something more profound. In this author's opinion, OOD embodies a unique, coherent theory of knowledge for system development. As such, it describes a cognitive process for capturing, organizing, and communicating the essential knowledge of the system's "problem space," and, for the model thereby formed, gives guidance on using specific techniques to map this problem space model to a "solution space" model.

The challenge of conceptual models has always been much more than just increasing the number of tools in the developers toolbox. To say "here is a bunch of tools to choose from" is very different from being able to say "here is a coherent theory of knowledge for system development *and* here is a bunch of tools to choose from that can be applied consistent with the theory." The theory gives guidance on what tool is most applicable for a particular situation. Herein lies a danger in the toolbox metaphor: it can be vulgarized to omit the critical need for a corresponding theory of knowledge on the use of the tools. It is nice for the carpenter to have many screwdrivers in the toolbox, but just as important is to *know* when it is appropriate to use the Phillips screwdriver and when to use the standard screwdriver.

Thus, a theory of knowledge is more comprehensive than a particular method for a particular life cycle phase. The work being done on object-oriented analysis (OOA)[1,4] signals the fact that OOD has gone from being a partial to a full life cycle approach, which means that OOD is much broader than just a "method." The emerging OOD paradigm promotes a particular mode of thought, a particular system viewpoint, which has far-reaching implications.

In the context of our discussion, a theory of knowledge for system development must give guidance on how to cope with complexity. Many of today's systems are exceedingly complex, and it is impossible to consider equally all of their attributes. Consequently, to understand such a system an *abstraction* must be used, whereby the most important attributes of the system are illuminated, and the others are suppressed. But, of course, this involves making assumptions about which aspects are more important, and which are less important. OOD is an abstraction that makes different assumptions than structured development (SD) about what to illuminate and what to suppress. Thus, OOD offers a unique and different perspective on what point of view best guides the thinking of the system modeler.

3. TOWARD A DEFINITION OF "OBJECT-ORIENTED"

3.1 Clearing the "Semantic Thicket"

Why is there so much confusion over what "object-oriented" means? To address this question, consider the origins of the term. "Object -oriented programming" emerged as a term associated with the development of Smalltalk, the original object-oriented language, and the one most often regarded as a pure object-oriented language.

Smalltalk drew heavily from Simula, ultimately incorporating the notion of *class* as the sole structural unit, with instances of classes or *objects*, being the concrete units which inhabit the world of the Smalltalk system. To a lesser degree, Smalltalk was influenced by LISP.

Rentsch has provided an interesting history of the origins of object-oriented programming and Smalltalk, and, in 1982, predicted the confusion that would accompany its growing popularity: "My guess is that object oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is."[2]

To get to the heart of the confusion surroundingf OOD, it may be helpful to consider the following points. First, OOD as a development philosophy has its roots in object-oriented *programming*, and evolved bottom up, i.e., from programming to design to requirements analysis. Thus, it is helpful to be familiar with object-oriented programming to grasp the OOD paradigm.

Second, OOD involves an unfamiliar way of thinking to most software developers, and in this respect some confusion is to be expected. Our training has not taught us to think in object-oriented terms. This dynamic can be seen when people are introduced to any new manner of thought. When the structured development techniques were first introduced, they also involved a new way of thinking about software development, and this led inevitably to some confusion. As another example, programmers often state that their first exposure to the simulation language GPSS V "blows their minds" because it involves a way of thinking that is radically different from the procedural languages they are used to.

Beyond these two factors, however, OOD has faced an even deeper problem in being understood. Early on in the history of object-oriented programming, bits and pieces of the concept were supported in various languages, and these languages became erroneously regarded as "object-oriented." Ada is the main example of this. As Ada was taking form in the late 1970's, it was widely understood that it was, in large part, based on features found in Pascal and ALGOL. Later, as Ada became touted as an object-oriented language, people who had experience with Pascal or ALGOL naturally made the connection in their minds that "object-oriented" was somehow a souped up

extension of a Pascal-type procedural language. As Cox said, in commenting on the confusion surrounding the object-oriented buzzword, "It is hard to imagine two languages more different than Smalltalk-80 and Ada, yet both are sometimes called object-oriented languages."[3]

In addition, probably due to the heritage of Simula, some modeling techniques that utilize classification have been advertised as being object-oriented. The book by Shlaer and Mellor, *Object-Oriented Systems Analysis*,[4] has been criticized by some as being little more than an approach to semantic data modeling.[5,1] (Interestingly, apart from its title, the term "object-oriented" hardly ever appears in the book.) However, while this author generally agrees with the above cited criticisms as to the incompleteness of the method presented, the book does provide useful guidance for the front-end activities of an object-oriented approach, specifically in the critical task of identifying the "objects," which will be discussed below.

So, object-oriented has become a term that means different things to different people. It is this author's opinion that a uniform definition of object-oriented is possible and desirable, at least in terms of the *characteristics* of an object-oriented approach. Moreover, the literature on the subject is completely adequate in this respect. In coalescing around a definition we should be primarily guided by the original intent of the term, especially in its connection to Smalltalk, and by the development of the theory that has built upon that original meaning, which extended the notion to design and analysis.

3.2 Comparing Development Paradigms

Structured analysis, structured design, and structured programming are collectively known as structured development (SD). The ideas behind these methods are found in the writings of Yourdon and Constantine,[19] Dijkstra,[20] DeMarco,[21] Myers,[22] and many others. Although there are several versions of SD in existence today, all of them are based on a philosophy of system development that analyzes the system from a functional point of view. In describing this viewpoint, Constantine says "the main features of software that are of interest to the analyst, designer, or programmer are the functional aspects: what functions or tasks the software must perform, what subfunctions or subtasks are needed to complete the overall functions, what pieces or component parts will perform various functions, how those functions will be performed."[23] Constantine calls this way of thinking the "functional paradigm." A paradigm is an exemplary model that also constitutes a world view.

Constantine points out that "this focus is so embedded in our thinking about computer programming that we are scarcely aware of it. We do not think of it as constituting a particular way of viewing programs or consider the possibility of alternative views."[23]

Given that this is the current conventional way of thinking, Constantine provides a

concise comparison of the functional and object-oriented viewpoints:

> In the functional paradigm, function and procedure are primary, data are only secondary. Functions and related data are either conceived of as independent, or data are associated with or attached to the functional components.
>
> The antithesis of this viewpoint is the object-oriented paradigm. In this paradigm, data are considered primary and procedures are secondary; functions are associated with related data. Problems and applications are looked at as consisting of interrelated classes of real objects characterized by their common attributes, the rules they obey, and the functions or operations defined on them. Software systems consist of structured collections of abstract data structures embodying those object classes that model the interrelated objects of the real-world problem.
>
> From this it should be clear that the object-oriented paradigm is based on a new way of looking at the old dichotomy of procedure and data. [23]

The SD techniques are generally associated with a top-down development strategy, whereas OOD is essentially a bottom-up approach. The "top" of a system structure contains control modules representing the activation of large chunks of the overall capabilities. At the "bottom" are the basic facilities for defining and manipulating the data, and for hiding its structure from the rest of the system. These concepts are embodied in the idea of the "object," as will be discussed later. OOD thus facilitates a natural bottom-up organization of the software.

3.3 Characterizing Object-Oriented Systems

With this overview of the two development paradigms as a framework, below are brief descriptions of the characteristics of object-oriented development. The following discussion is not meant to be a tutorial on the subject, but is presented to help formulate a definition. For more complete discussions of specific object-oriented methods, the reader is referred to the references, especially Coad,[1] Booch,[7] Cox,[3] Rentsch,[2] and Meyer.[26]

Objects

At the heart of OOD is the idea of the "object." An object is an entity defined by a set of common attributes, and the services or operations associated with it. Objects are the major actors, agents, and servers in the problem space of the system, and can be identified by carefully analyzing this domain. Whatever form of system requirements that is provided by the user or client can be used as a starting point for this task. Objects can be found among devices that the system interacts with (e.g., sensors), other systems that interface with the system under study, organizational units (departments, divisions, etc.), things that must be remembered over time (e.g., details about events occuring in the system's environment), physical locations or sites, and specific roles played by humans (clerk, pilot, etc.).

The central problem in OOD turns out to be "finding the objects." As mentioned above, Shlaer and Mellor have made a contribution in this area, providing tools and concepts

for enumerating various categories of potential object classes.[4]

In characterizing the object concept it is important to discuss the viewpoint associated with objects. OOD is interested in how an object appears (an outside view), rather than what an object is (an inside view). This shift in viewpoint is critical to the concept of OOD. It enables the system modeler to think not only about objects as classification entities, but also about the intrinsic behavior associated with objects. This intrinsic behavior embodied within an object provides the visibility of the object in relation to other objects.

For example, consider the object called "policeman." The object called "ordinary citizen" is not much interested in exactly what a policeman *is* (inside view) but is *very* interested in the *intrinsic behavior* (outside view) of the object (issues speeding tickets,etc.) Also the ordinary citizen object is interested in what *services* it can invoke from policeman (give directions when lost, etc.), which are also part of the intrinsic behavior of the object.

The term *encapsulation* has come to represent this central notion of OOD, and is based on the concept of *information hiding*, as espoused by David Parnas.[27] Rentsch stresses the importance of this viewpoint for OOD, saying

> The shift in viewpoint from inside to outside is itself an essential part of object oriented programming. In my experience this shift occurs as a quantum leap, the "aha!" that accompanies a flash of insight. In spite of this, I am convinced that the view from outside is the natural one - only my years of training and experience as a programmer conditioned me to "normally" view objects from inside ... The first principle of object oriented programming might be called *intelligence encapsulation:* view objects from outside to provide a natural metaphor of intrinsic behavior.[2]

Nonhierarchic Object Model

Developers accustomed to using structured techniques usually think of the system-structure model as a hierarchy of functions, which maps to a set of nested subroutines. (It should be noted that, at least in the case of structured design, this is a manifestation of the way the method is practiced, not in the method itself. Yourdon and Constantine explicitly allowed for nonhierarchic module collections, as discussed in chapter 18 of their book, *Structured Design* .[19]) The object model, on the other hand, is a nonhierarchic topology of objects. This topology forms an abstract view of the problem space, whch is meant to map naturally to a nonhierarchic model of the solution space. As Booch says, "a program that implements a model of reality may be viewed as a set of objects that interact with each other."[7]

Processing and Communication

In the OOD model, processing takes place inside objects. "An object, far from being inert matter, is an active, alive, intelligent entity, and is responsible for providing its own computational behavior. Thus processing capability is not only inside the object, it is everpresent within and inseparable from the object."[2]

While an object contains processing capability, it is not always self sufficient in this respect, and may need to interact with other objects. The "ordinary citizen" object, for example, can do many things for itself (eat, read, write), but may at times need to invoke the services of the "policeman" object (get directions). Thus communication between objects is needed.

The model being nonhierarchic, objects communicate with other objects by a mechanism of "message passing," which is similar to the concept of "coroutines" as discussed by Yourdon and Constantine.[19] Thus, an invocation of one module by another is not necessarily associated with a hierarchical relationship in the organization of the objects. "Message sending serves as the uniform metaphor for communication in the same way that objects serve as the uniform metaphor for processing capability and synthesis. ... all processing is accomplished by message sending. ... A message serves to initiate processing and request information."[2]

Inheritance

Basic to the idea of OOD is a framework of classing, subclassing, and superclassing, that allows individuals within a collection to *share* common attributes, as needed. This framework is collectively referred to as *inheritance.* If the class "tree," for example,, contains the attribute "leafy branches," then members of the tree subclasses (oak, elm, maple, etc.) can inherit this attribute, and it need not be redefined.

Object-Oriented Software Organization

There are several variations of object-oriented methods, all of them being somewhat sketchy. Booch, for example, has offered the skeleton of a step-by-step approach for object-oriented design,[7] and other writers have made similar gestures. Consequently, it is impossible to describe *the* object-oriented method, or to be very specific in discussing exactly what *the* end product of OOD looks like. However, it is possible to discuss a general object-oriented concept that refers to a way of structuring software systems that is language independent.

Object-oriented software organization refers to a structuring approach that makes it possible to organize a system by object-oriented concepts and implement it as a set of "object modules" in conventional procedural languages as well as in object-oriented languages. Constantine has this to say about this concept:

> Object-oriented organization is a form of modular structure in which a program is designed as a structured collection of "object modules" implementing data abstractions. An object module encapsulates the implementation of a class of data structures and their related services, exporting specific services (and sometimes attributes) as features of the "official" external interface of the module. Object modules are interrelated by their usage of the attributes and services of other modules, just as in conventional functional structures, as well as by feature inheritance. An object module is thus an organizing concept that groups together a collection of functions (services) based on their association with a well-defined class of data structures. Object modules are, in effect, multi-function packages that group functions based on principles of strong association with common data structures, with groups interrelated and defined as classes and subclasses of abstract data. [23]

3.4 Deciding if an Approach is Truly Object-Oriented

The preceeding discussion can be summarized to say that an object-oriented method contains at least these basic ideas: *classification* of data abstractions, *inheritance* of common attributes, and *encapsulation* of attributes, operations and services.
If these characteristics form the basis for a definition of "object-oriented," then it is clear that some languages and methods which are often cited as being object-oriented are something less. According to Peter Coad and Edward Yourdon:

> One can look at different languages, environments, methods, and books and ask "Are you really Object-Oriented?" Unfortunately, many times the answer is "no"; Object-Oriented suffers from being a catchy marketing word, used at times to mean "the good stuff."
>
> Is Ada an Object-Oriented language? No. Genericity, that is, typed parameters, is convenient, but no substitute for inheritance.
>
> Is an information modeling approach Object-Oriented (e.g., Object-Oriented Systems Analysis [Shlaer and Mellor, 1988], a book better titled "making Semantic Data Modeling Practical")? No. Services are missing. Classification is missing. Inheritance is missing. [1]

Coad and Yourdon offer this equation for recognizing an object-oriented approach:

> Object-Oriented =
> Objects
> (an encapsulation of Attributes and exclusive Services;
> an abstraction of the something in the problem space, with
> some number of instances in the problem space)
> + Classification & Inheritance.[1]

The importance of a clear, consistent definition of OOD is that the *essence* of this paradigm for system development not be watered-down, or completely lost in the shuffle, by a loose use of the term. As discussed earlier, OOD represents a theory of knowledge, a mode of thought, that is decidedly unique in its own right. Our use of the term should be directed toward that uniqueness, not toward certain specific *features* of OOD. However, this does not mean that languages or methods that contain certain of those features are of no use. Many of them are, but we should be careful not to characterize them as being "object-oriented," lest we continue to foster the confusion over the term.

4. IS OOD BETTER THAN SD?

A great deal of hopeful enthusiasm has accompanied many of the writings on OOD. According to Cox, "It is time for a revolution in how we build systems ... The revolution is object-oriented programming."[3] From Booch, "[OOD] leads to improved maintainability and understandability of systems."[7] From Danforth and Tomlinson, "[object-oriented programming] represents a positive step toward the design and

implementation of complex software systems."[8]

The proponents of OOD usually cite two reasons for their excitement about the approach. One is the claim that the thinking process inherent in OOD is more "natural" than that of SD, i.e., in building an abstract model of reality it is more natural to think in terms of objects than in terms of functions. The other is that the modeling of the problem space maps more directly to the solution space in OOD than it does in SD.

Let us consider the latter argument first. Anyone involved with SD knows that the transition from the analysis model to the design model can be tricky. For example, in moving from a data flow diagram view of the system to creating design structure charts the modeler is forced to make a significant shift in perspective. There are strategies to assist in the matter (transform analysis, transaction analysis, etc.), but it remains a difficult task because the mapping is not truly isomorphic. The proponents of OOD often claim that with OOD the mapping *is* truly isomorphic, or at least far more isomorphic than in SD, which, if true, would offer a great advantage. Coad and Yourdon make this claim, and stress that the singular abstraction medium is a key: "It's a matter of using the same underlying representation in analysis and design. This is a bedrock concept of the [object-oriented analysis] approach."[1]

But is this claim true? Unfortunately, there is little evidence of this in the writings on the subject, a fact which, in part, reflects the relative newness of the field. In fact there are those that refute the notion completely. Constantine says "The mapping from analysis model to design model is every bit as 'not isomorphic' in OOD as in SD once you get beyond textbook and toy examples" (his emphasis).[24] It seems the jury is still out on this question, and the matter will not be resolved until there is some published hard evidence.

Part of the difficulty in resolving this matter is that the overall development environment seems to be largely inadequate to exploit the paradigm fully. Stroustrup has compared several languages in terms of their support for OOD, and points out that if the proper language support is missing the paradigm will be of limited value.[9] And Williams says, "Due to the informality of the approach, lack of adequate language support, and lack of experience with large systems, the potential of the object model in software engineering remains largely unexplored."[6] As might be expected, there is much on-going research in this area.[10] But, the potential emergence of a more isomorphic mapping of problem space to solution space in the OOD paradigm is itself an important reason for software professionals to stay abreast of the developments in the OOD world.

Now to the argument that thinking in terms of objects is more "natural" than thinking in terms of functions. As cited earlier, Rentsch is "convinced that the [OOD] view is the natural one," and as evidence he posits that "children learn Smalltalk very quickly."[2] Coad and Yourdon state that object-oriented analysis "is based upon concepts which we first learned in kindergarten: objects and attributes, classes and members, wholes and parts." They further assert that for today's modern systems "an object-oriented

approach... is a more natural way of dealing with such systems."[1]

It is enticing to think that we have stumbled upon a more natural way of thinking after all these years, but does the evidence really support this notion? It is true that kindergarten children readily learn to classify objects, and it may be true that they learn Smalltalk quickly. But they *also* readily ("naturally?") think in terms of functions, and they learn function-oriented procedural languages quickly. Ask a child to tell you about his or her school, and then try to predict if the response will be primarily in functional terms, or in classification terms (but don't bet on it being one or the other). Is one way of thinking really more "natural" than the other?

Constantine doesn't think so, saying, "there are no 'object classes' in the real physical universe, as these are constructs that exist only in the mind of the viewer... 'objects' are no more real or natural than 'functions.'"[23]

Interestingly, this same type of debate exists in other fields of study. In fact some variation of the "form vs. function" controversy is part of every field. In general, it involves how one views the relative importance of the structural vs. the functional aspects of knowledge. In developmental psychology it is the debate between Piaget and the Gestaltists, and then later between Piaget and the new functionalists;[11,12] in epistemology the debate focuses on the relation between the knower and the object known ;[13] in cognitive anthropology the issue revolves around how a grouping of people synthesizes structure and function as they form their own model of reality in pursuing an understanding of the world they live in.

Two Anthropologists, Dr. George Brandon of the University of Maryland, and Dr. Elliot Fratkin of Duke University were interviewed by the author for the purposes of this paper. Both of them were intrigued that such an issue was being argued in the software field, and saw the clear connection to the debate in their own discipline. Both believe it is a manifestation of a universal question, and were decisive in their opinions that the matter is not resolved, and probably never will be resolved.

Obviously, the details of these various debates is beyond the scope of this paper; but the important finding is that there is no final resolution to the matter. The struggle is ongoing. There is very little that you can say with much confidence about a "most natural" way that people think about the realities of their universe. Thus, to say that the object model is a more natural way to think involves a rather sizeable leap of faith.

Undoubtedly, the truth of the matter is that both paradigms are "natural," and that the proper synthesis of the two, in relation to a particular problem, is what should be striven for. In fact, this is part of the challenge for the next period. OOD has helped to crystallize a way of analyzing systems that brings into sharp relief the fact that we have been limited and one-sided in our approach to the matter up to this time. But we must be careful not to throw the baby out with the bath water. Functional partitioning is a very valuable analytical approach. The task ahead is to move the debate to a higher level - not arguing about which is more "natural" - but exploring how we best take

advantage of both approaches.

In this author's view, Ward and Mellor were on the right track when they suggested that the system modeler must determine which modeling approach should play a dominant role, based on the particular type of system being modeled.[14] As the methods mature, guidelines need to be developed and refined to help the modeler determine whether the perspective of OOD or SD should dominate in a particular situation.

5. TOWARD AN INTEGRATION OF THE METHODS

Over the past few months, several papers have addressed the synthesis question mentioned above. Ward has recently written a brief tutorial showing that there is no inherent conflict between the two approaches, and that "real-time structured analysis / structured design can, with modest extensions to the notation and to the model-building heuristics, adequately express an object-oriented design."[15] Jalote proposes an "extended object-oriented design methodology," which incorporates a top-down, step-wise refinement approach.[16] Bailin describes a method for combining structured analysis with the object-oriented approach for requirements specifications.[17] Constantine has recently written two papers that address the topic of the integration of the methods.[23,25] In one he stresses that we must get beyond the "madness of methods," and get "back to basics" by agreeing on a set of fundamental principles independent of methods. With sound principles being recognized as more important than specific methods the groundwork will be laid for a coherent integration of the methods. He points out that there are "a number of common principles [that] are revered by almost all the sundry gurus and acolytes," which can form a basis for what he calls "convergent design methods."[25]

This type of work is highly significant on two counts. First, it is useful to begin exploring these areas of compatibility for purely pragmatic reasons: contractual and/or documentation requirements may make it necessary. Second, and more important, it pushes the theory of knowledge for system development to a higher level, and facilitates the evolution of more coherent and useable methods.

6. RECOMMENDATIONS FOR PROJECT LEADERS

Every software development organization should begin figuring out where they stand in relation to OOD and SD. For shops already engaged with the structured methods, this means having a *plan* for assessing if, and how, OOD might be used to advantage. Ideally, such a plan should be part of a technology transfer program of the organization, as proposed by Loy,[18] and should proceed with both bottom-up and town-up activities being done concurrently.

A suggested plan outline is the following:

1. Have at least two, and preferably three or more, senior software engineers start playing around with Smalltalk. If Smalltalk is not an available option, pick another

language that incorporates the features mentioned in section 3 above. It is important that some experienced people begin learning the paradigm at the programming level by getting experience with an object-oriented language. This activity can be done part-time in conjunction with other ongoing assignments. But they should keep at it on a regular basis.

2. Concurrent with the above task, start tracking the literature on the subject. Have an organized effort to do this, with different people assigned specific responsibilities in this respect. For example, one person might be assigned to monitor the articles that appear in one periodical. Much has been written in the past year, especially, and the trend will undoubtedly increase. Continually assess how the experience in other shops relates to your situation. The references cited in section 5 above could be a good starting point.

3. After a few weeks, provide some of the folks involved in the above two tasks with some "formal" OOD training, e.g., a seminar in OOD. This should help to quickly broaden their perspective on the potential of OOD.

4. Get continual, systematic feedback from all people who are involved in the above tasks about their views on OOD. Do they feel it might play a role in your environment? How significant does its value appear to be? How should it be incorporated into your development plans? Their views on these matters may change over time, so it is important that the feedback mechanism be regular and systematic.

5. Based on the results of the above activities, begin to formulate options for integrating OOD into your environment. This will involve considering the whole spectrum of trade-offs specific to your situation.

7. SUMMARY AND CONCLUSIONS

Object-oriented development is bound to have a major influence on the manner in which systems are built. It has matured to where it must be viewed as a theory of knowledge that embodies a unique perspective for capturing, organizing, and communicating the essential features of a system. The OOD paradigm is derived ultimately from object-oriented programming, an understanding of which is helpful to grasp the essence of OOD.

The software engineering community has been sloppy in its use of the term "object-oriented," and there is currently confusion over the meaning of it. The term should refer collectively to the concepts of *classification, encapsulation,* and *inheritance.*

While some contend that OOD involves a more "natural" way to think than does SD, there is little evidence of this. In fact, both approaches are based on modes of thought that might be called "natural." OOD and SD are not incompatible, and much recent work has been devoted to exploring the potential synthesis and integration of the two.

Research in this area should help crystallize the contribution of OOD to our field, and push the quest for a theory of knowledge for systems development to a higher level.

Project leaders should have a plan for investigating the worth of OOD to their shops. This plan, a sample of which was offered, should be part of an ongoing technology transfer program.

Acknowledgements

Many people reviewed the first draft of this paper and provided very helpful, constructive criticisms. Many thanks are due, especially, to Larry Constantine, Jack Bond, Merlin Dorfman, Sally Shlaer, Stephen Mellor, and Bruce Blum for their insightful feedback.

REFERENCES

1 Peter Coad and Edward Yourdon, "An Introduction to OOA - Object-Oriented Analysis," (May 30, 1989), *Tutorial: Systems and Software Requirements Engineering,* Vol. 1, edited by Merlin Dorfman and Richard Thayer, IEEE Press, 1989.

2 Tim Rentsch, "Object Oriented Programming," *SIGPLAN Notices*, Vol.17, No. 9, pp. 51-57, 1982.

3 Brad Cox, *Object-Oriented Programming,* Addison-Wesley, 1986.

4 Sally Shlaer and Stephen Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, 1988.

5 T.L. (Frank) Pappas, "Object-oriented analysis is flawed," *IEEE Software*, January, 1989.

6 Lloyd Williams, *The Object Model in Software Engineering*, Software Engineering Research, 1986.

7 Grady Booch, "Object-Oriented Development," *IEEE Transactions on Software Engineering*, February, 1986.

8 Scott Danforth and Chris Tomlinson, *Type Theories and Object-Oriented Programming,* ACM Computing Surveys, March, 1988.

9 Bjarne Stroustrup, "What is Object-Oriented Programming?", *IEEE Software,* May 1988.

10 *Research Directions in Object-Oriented Programming*, edited by Bruce Shriver and Peter Wegner, MIT Press, 1987.

11 Herbert Ginsburg and Sylvia Opper, *Piaget's Theory of Intellectual Development*, Prentice-Hall, 1969.

12 *New Trends in Conceptual Representation: Chalenges to Piaget's Theory?*, edited by Ellin Kofsky Scholnick, Lawrence Erlbaum Associates, 1983.

13 *Academic American Encyclopedia*, vol. 7, 1981.

14 Paul Ward and Stephen Mellor, *Structured Development for Real-Time Systems*, Yourdon Press, 1985.

15 Paul Ward, "How to Integrate Object Orientation with Structured analysis and Design," *IEEE Software*, March, 1989.
16 Pankaj Jalote, "Functional Refinements and Nested Objects for Object-Oriented Design," *IEEE Transactions on Software Engineering*, March, 1989.
17 Sidney Bailin, "An Object-Oriented Requirements Specification Method," *Communciations of the ACM*, May, 1989.
18 Patrick Loy, "Five Components of a Software Quality Assurance Paradigm", *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, September, 1988.
19 Edward Yourdon and Larry L. Constantine, *Structured Design*, Yourdon Press, 1975.
20 E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.
21 Tom DeMarco, *Structured Analysis and System Specification*, Yourdon Press, 1978.
22 Glenford J. Myers, *Composite/Structured Design*, Van Nostrand Reinhold , 1978.
23 Larry L. Constantine, Object-Oriented and Structured Methods: Toward Integration, American Programmer, vol. 2, no. 7/8, August, 1989.
24 Larry L. Constantine, letter to the author (unpublished), July3, 1989.
25 Larry L. Constantine, "Beyond the Madness of Methods: System Structure Modeling and Convergent Design," *Software Development '89: Proceedings*, Miller-Freeman Publishing Co., 1989.
26 Meyer, Bertrand, *Object-Oriented Software Construction*, Prentice-Hall, 1988.
27 Parnas, David, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, December, 1972.

Constructing Reusable Software Using Feature Contexts

C. Frederick Hart*
John J. Shilling†
Georgia Institute of Technology

July 30, 1989

Keywords: Software Reuse, Software Maintenance, Program Slicing, Software Engineering

Abstract: Explicit documentation of software features, using feature contexts, provides an effective way to create easily customized software components thus promoting reuse and enhancing maintainability.

Frederick Hart is currently working toward a Ph.D. in Information and Computer Science at the Georgia Institute of Technology where he also received his B.S. in Physics and M.S. in Information and Computer Science. From 1984 to 1987 he was employed by NCR Corp. working in the areas of data communication and compilers. For the past two years he has been a research assistant with the Georgia Tech Research Institute's Artificial Intelligence Branch working on expert systems and software reuse. His research interests include software engineering environments, software reusability, and formal program development techniques. Frederick Hart is a member of the ACM, AAAI, and IEEE Computer Society.

John Shilling received a B.S. degree in Mathematics in 1982 from The University of Texas at Austin, and M.S. and Ph.D. degrees in Computer Science in 1984 and 1986 from the University of Illinois at Urbana-Champaign. He performed two years of post-doctoral research at the IBM T.J. Watson Research Center before joining the faculty of the Georgia Institute of Technology in the fall of 1988. His research concerns the structure and algorithms of software development environments. He helped develop the Fred and RPDE structure-based environments and developed a prototype automated reference librarian system that is accessed as an integrated part of software construction. His current work involves the logical views approach to software development environment architecture. The approach extends the object-oriented paradigm of object-base architecture to allow powerful, but controlled, interaction between related structures. His work also includes extending the internal representation of structure-based environments to facilitate the construction and manipulation of parallel programs. Professor Shilling is a member of ACM and IEEE.

*Georgia Institute of Technology, PO box 37174, Atlanta, GA 30332, (404) 894-3309, cfh@pyr.gatech.edu
†School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332-0280, (404) 894-3807, shilling@gatech.edu

Constructing Reusable Software Using Feature Contexts

1 Introduction

This paper draws together two seemingly disparate aspects of software quality, software maintenance and software reuse, under the common theme of modifiability. Modifiability is clearly an issue in software maintenance. Accurate and timely maintenance of software has long been recognized as one of the more difficult aspects of software engineering. Maintenance of software appears to obey the second law of thermodynamics: the entropy of a maintained program seems never to decrease. The more maintenance is performed, the more disorder increases and the harder to modify the software becomes. Much of the maintenance problem can be attributed to the ad hoc manner in which it is generally performed. For all the work in structured programming, comparatively little attention has been paid to the concept of structured maintenance. Methods that are adequate on a small scale become disastrous as systems become more complex.

Modifiability is also an issue in software reuse because reusable components often need to be modified to some degree[1] prior to their inclusion in a new piece of software. The need for modification results in conflicting criteria for reusable components. On the one hand, the more complex a component is, the greater the profitability from reusing it. However, such complex components tend to be more specific and therefore need more modification in order to be usable in new situations. Once again, modifiability plays an important role.

An important factor in the construction of easily modified software is complete and accurate design documentation. Numerous techniques have been developed for documenting program design and implementation. Unfortunately these techniques do not address the problem of dependencies and correspondences between the design documentation and implementation. For example, suppose the design of an interactive application specifies a *help* function to provide information on available commands. The implementation then contains sections of code (not necessarily in one location) which implement the specified help function. There is then a correspondence between the specification of this help function and the (possibly distributed) code which implements it. In addition, the information provided by the help function for a given command depends both on the body of code implementing that command and on the method of activating that command. This means that if either the name used to invoke the command or the code performing the command changes then the information in the help function needs to be updated.

In current methodologies and support environments, these dependencies and correspondences within the specifications (documentation) are for the most part only implicitly represented. In the rare instance in which such a condition is mentioned explicitly, it carries only the force of a comment which may be ignored or overlooked. As a result, the maintainability of the software suffers. In addition, as the software evolves, dependencies, correspondences and whole features of the software are added, deleted, and modified. Often modifications made to the code are not accurately reflected at higher levels. This in turn aggravates the difficulty of subsequent maintenance of the software.

What is needed is a methodology and environment which treats all aspects of an evolving software system in a unified manner. This means that the end product of such a system is not a group

[1] This is often referred to as the *same as except* problem - meaning a component is needed which is exactly like an available one except ...

of loosely related design documents and source code modules but a single integrated product. Requirements, design and source code can then be thought of as different *views* of a single entity. Through automated maintenance of explicit design dependencies and correspondences, the problem of unrecognized relationships in the design and code is mitigated and accurate update of design documentation during maintenance is facilitated. Links maintained by such a system would allow it to answer such questions as "What code implements this feature?" or "What function is this code performing or supporting?".

2 Linking and Feature Contexts

In light of the preceding discussion, the development of an environment and associated methodology to support an explicit fine-grained linkage of design specifications and implementation is under way. The system is to be based upon an object-oriented design methodology. This paradigm provides for logical organization of designs and will support well defined interfaces within the designs simplifying construction, understanding, and linking of the specifications and implementation. It will be supported by a program construction/editing environment which will provide for the establishment and maintenance of links between the various portions of the design and implementation. Slices or views of programs developed using this environment can then be examined or modified - with the environment assuming the responsibility for maintaining the different views.

2.1 Linked Structures

As mentioned above, understanding and customizing retrieved components is one of the major tasks facing the user of any library of software components. The environment described in this paper will assist with these tasks in two ways. First, the linking of designs with implementations will provide for more explicit documentation of design decisions and implementation methods. This assists in understanding of the component and thus in its modification. Second, the linking supported by the system can be used to isolate individual *features* of the component into *feature contexts*. *Features* can be thought of as logical units of program functionality. They are important because they often correspond to units of change during maintenance. *Feature contexts* are simply the realization of a feature in the environment. Feature contexts consist of design documentation describing a feature, the code implementing the feature and, most importantly, the environment maintained information which delineates the feature's implementation and links it to the design documentation. As a result of the linking, components can be incrementally constructed a-feature-at-a-time with the design of each feature and its associated implementation clearly isolated from the rest of its host component. Feature contexts facilitate the automated configuration of a component from the design level via removal or addition of features associated with the component. Features also provide for the localization of software functionality. As has often been noted, functions which are documented in one place in the design of a program are often spread out across the implementation. This results in a delocalized implementation of the function and greatly increases the difficulty in maintaining the software. Feature contexts ameliorate the maintenance problem by providing enhanced documentation of delocalized features and mechanisms for manipulating their distributed implementations.

2.2 The Nature of Designs

The concept of features and feature contexts is based on the idea of linking the design of a program or component with its implementation. In order for this linking to be effective, several things are required of the designs. Most importantly with respect to the work described in this paper, the designs must support the definition and manipulation of features. To do this they must provide structures for defining features, handles for accessing the linked implementation, and help in insuring the consistency of the linking. The designs must also document the structural decomposition of the implementation along with providing user-oriented textual descriptions of each design element. Finally, since one of the goals of the this system is to provide configurability for reuse, the design must provide mechanisms for easy integration of components.

In consideration of these requirements, software designs constructed for use in this system form an object oriented class hierarchy employing multiple inheritance. Designs are subdivided into nodes. The nodes of a design are linked together via references contained within the body of each node. For example, a component node might contain links to other inherited components, defined methods, and component features (see figure 1). These links can be used to navigate around the design in a hypertext like fashion. In addition to links between design elements, there are links to the implementation corresponding to a given design element. This form of documentation facilitates rapid perusal of the documentation of distributed software features and also provides a mechanism for performing the automated configuration of component implementations mentioned above and described further in the next two sections.

There are four major types of design nodes. First there are *component* nodes. These correspond to class definitions in an object oriented language. Component nodes describe the internal state and the public and private interface of a given component. Each of the interface routines are further developed in *method* nodes. There is a method node for each method of a component. Method nodes informally describe the function performed by the method and trace the process structure of the method. The third type of design node is the *feature* node. Feature nodes can be children of either component, method, or other feature nodes. These nodes describe the structure and function of the given features. Proper feature definition is the key to the effective use of the system described in this paper. Finally, there are *dependency* nodes which make explicit dependency relationships between implementation and design elements. Dependency nodes can also be children of component, method, and feature nodes. Dependency nodes are generally used to help insure the consistency of the design and implementation linkage. More will be said about the content of feature and dependency nodes later in the paper.

The ability to easily modify and configure software afforded by this approach is based on the definition of software features in the designs. There are no hard and fast rules which define what is and is not a feature. Indeed it is this flexibility in designing features which contributes to the power of the approach. The determination of the proper feature set of a component or application is quite similar to the problem of proper class design in object oriented systems. Although there is no strict formula for solving the latter problem there are intuitive guidelines which produce good results. The same is true for feature design. The most basic guiding principle for designing features is that they form a self contained unit of functional meaning. This generally means that the feature can be concisely described in terms of its function or properties it possesses. Examples of a features which fit this definition might be the error checking code in a component or the operating system dependent code in an application.

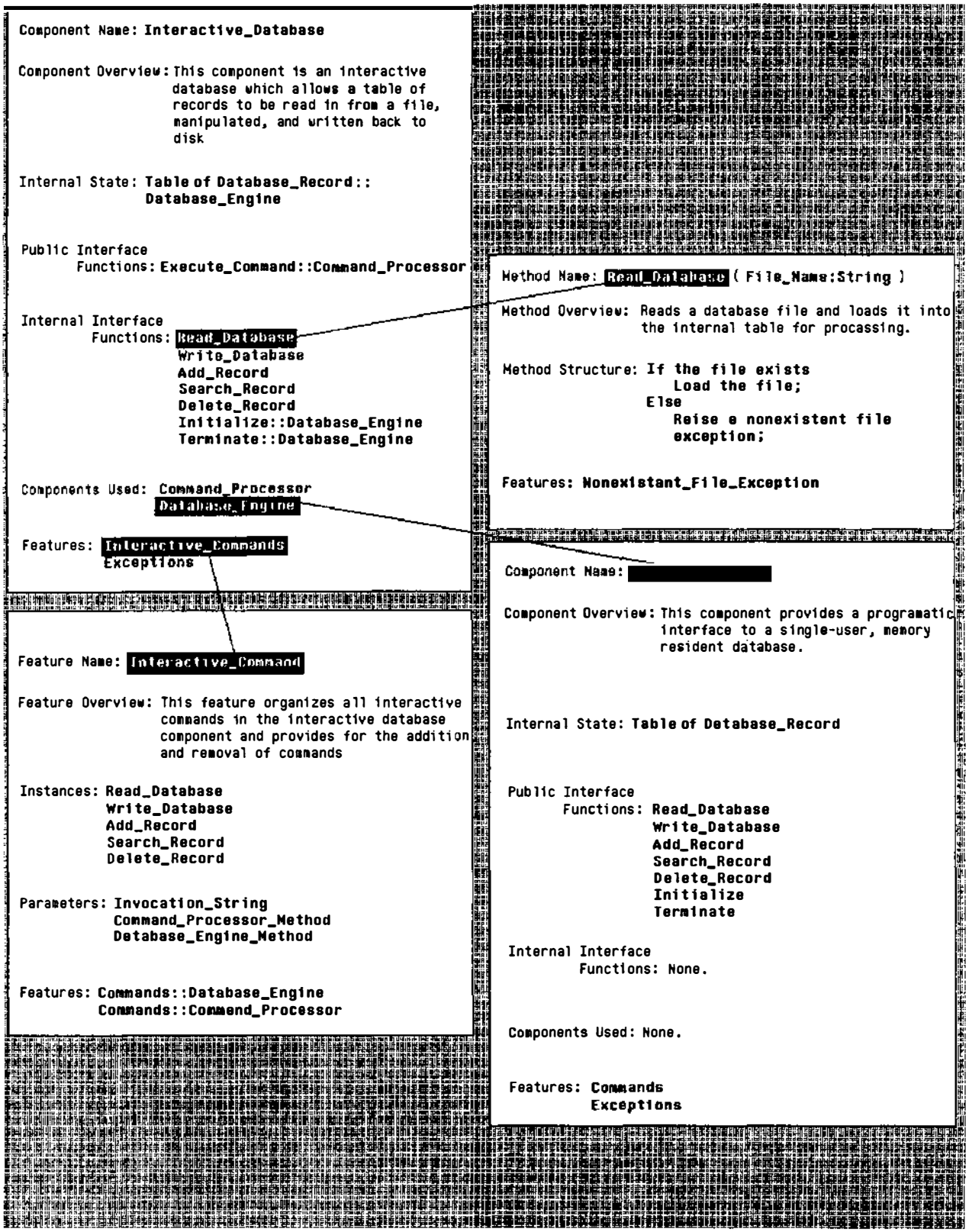

Component Name: Interactive_Database
Component Overview: This component is an interactive database which allows a table of records to be read in from a file, manipulated, and written back to disk
Internal State: Table of Database_Record::
Database_Engine
Public Interface
Functions: Execute_Command::Command_Processor
Internal Interface
Functions: Read_Database
Write_Database
Add_Record
Search_Record
Delete_Record
Initialize::Database_Engine
Terminate::Database_Engine
Components Used: Command_Processor
Database_Engine
Features: Interactive_Commands
Exceptions
Method Name: Read_Database (File_Name:String)
Method Overview: Reads a database file and loads it into the internal table for processing.
Method Structure: If the file exists
Load the file;
Else
Raise a nonexistent file exception;
Features: Nonexistant_File_Exception
Component Name:
Component Overview: This component provides a programatic interface to a single-user, memory resident database.
Internal State: Table of Database_Record
Public Interface
Functions: Read_Database
Write_Database
Add_Record
Search_Record
Delete_Record
Initialize
Terminate
Internal Interface
Functions: None.
Components Used: None.
Features: Commands
Exceptions
Feature Name: Interactive_Command
Feature Overview: This feature organizes all interactive commands in the interactive database component and provides for the addition and removal of commands
Instances: Read_Database
Write_Database
Add_Record
Search_Record
Delete_Record
Parameters: Invocation_String
Command_Processor_Method
Database_Engine_Method
Features: Commands::Database_Engine
Commands::Command_Processor

Figure 1:

The former performs a specific function whereas the latter may perform many different functions but all instances share the property that they are OS dependent. A second common attribute of features is that their implementations are often not localized. The ability to bring together in one place the implementation of a distributed feature of the code is an important property of feature contexts. Traditional specifications and implementations allow only a single decomposition of a software system. In general this decomposition gives rise to the module and procedure hierarchy of the system. What happens when it is desirable to take a functional slice of a system which does not correspond to the current modular decomposition of the system? Features** allow these secondary decompositions to be explicitly recorded and their implementations localized in the design. Furthermore, if it is desirable to vary the properties of a software system by adding or removing these feature slices, feature contexts provide a mechanism to do so.

2.3 Feature Extraction

The linking of designs to implementations simplifies the creation of easily modified components by providing mechanisms for defining variations. The principal mechanism by which variations are organized is the feature context. Grady Booche, for example, describes a number of feature variations for a stack package [1]. Options include iterator/noniterator, concurrent/single-user, etc. Each of these can be viewed as an optional addition to a stack object much as optional features can be added to an automobile. To a simple stack one can add an iterator, concurrent access capability, or memory management. A library component can then be designed and constructed which contained all the available options, each isolated into a separate feature context (see figure 2). A component which includes a subset of the available options can be quickly derived from a component containing all variations by specifying the desired feature contexts. This allows the component author to construct a single full-featured component for the library instead of a set of all possible variations of the component. The ability to extract features means that component enhancements and bug fixes need to be made in only one component and not a family of components as would be necessary without the extraction capability. Feature contexts also facilitate evolutionary development of components since a user, not finding a desired feature, can add a new feature context to that one component and resubmit the component to the library. Existing applications are not affected by the code change, since the change is isolated into its own feature context which will not be listed in their extraction list when deriving an implementation based on the now modified component. The linked design helps with the addition of new features in several ways. Linking fosters quicker understanding of the code, location of existing feature implementations, and helps insure accurate correspondence between the code and the design documentation. All this aids a programmer, especially one not familiar with the component, in implementing enhancements to the component. This linking is key since it provides for the evolutionary enhancement of the component which is necessary since a component designer can never anticipate all the variations of a component that may be needed.

Feature extraction is similar in many ways to the data and control flow based slicing done in [2]. One definition of program slicing used in the preceding paper defined a program slice with respect to a set of variables as the minimal subset of a program's code such that the values taken on by the variable set during execution of the slice and the original program on any given data set are identical. Similarly, if a component with a subset of available features is extracted from a full featured component, the result should be a minimal program capable of implementing the desired feature set. Of course unlike variable subsets, arbitrary feature subsets are not always possible

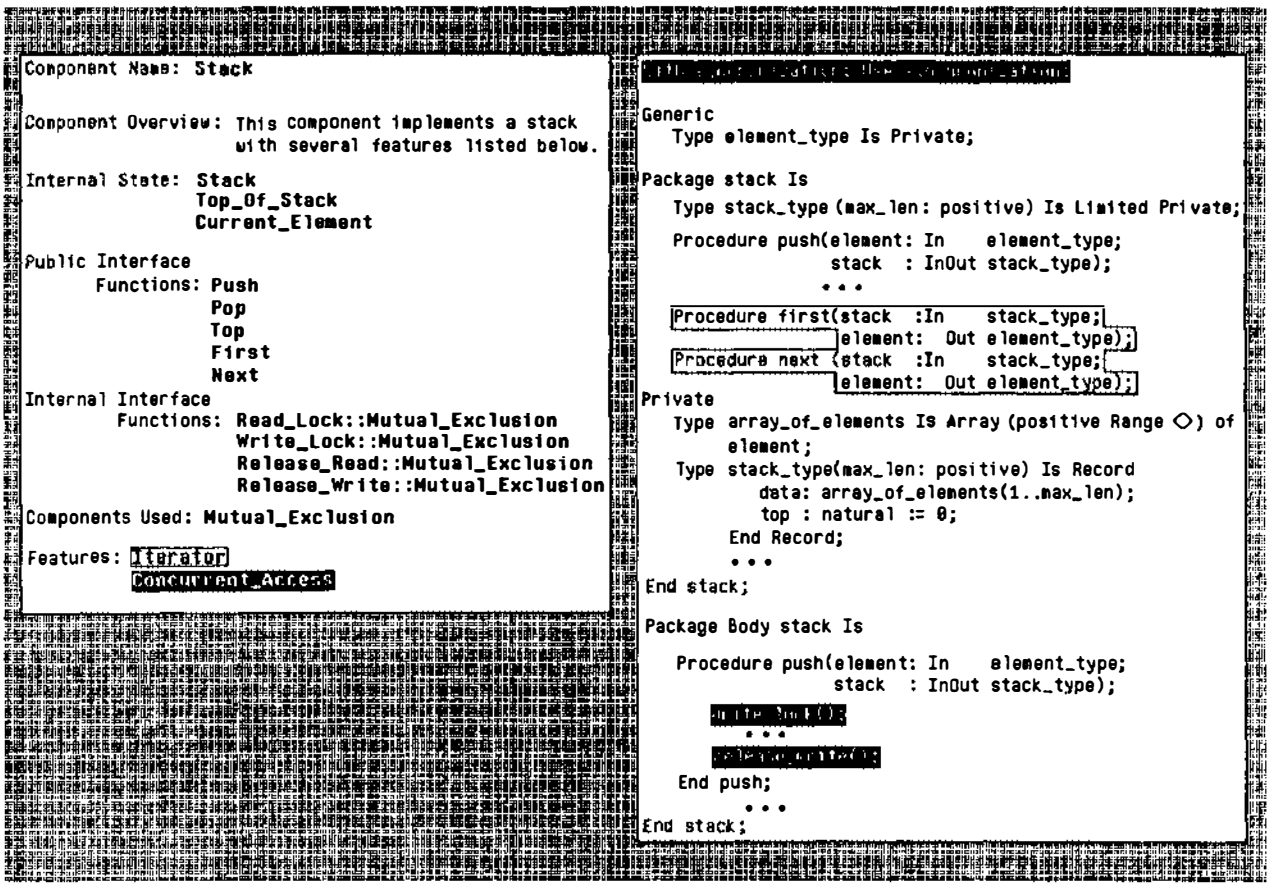


Figure 2:

since a given feature can depend on other features and therefore the dependent feature cannot be present without the supporting feature. However, many examples of orthogonal feature sets within existing application components have been constructed demonstrating empirical evidence for the viability and usefulness of feature extraction as a method of configuring software components and entire applications.

2.4 Generic Features

Feature contexts, in addition to organizing features for extraction, provide organization for generic features. Design elements in a feature context can be parameterized to form generic features. These generic features allow for creation of operations like the adding and deleting of button instances in a windowing component. Of particular importance, is the fact that feature contexts can link separate portions of the program into a single generic unit and can provide for generic *additions* to, or actively guide extension of existing code as opposed to simply providing the inactive building blocks of language based generic modules. Dependency information can also be included which helps to insure or automate the instantiation of generic parameters. Generic feature contexts share all the capabilities of traditional generic modules (i.e. Ada Generics) and extend their capabilities in important ways. This extra power is derived from the ability of the proposed environment to maintain multiple simultaneous design views of an object (system, program, module collection etc.)

under consideration. Additionally, the fact that these generic abstractions are at the level of design and not at the level of the implementation language's syntax enable generic constructs to span syntactic boundaries that would otherwise limit their use.

2.5 Significance of Feature Contexts

An important influence on the development of the concept of feature contexts was the research of Vasant and Jarke in [3]. The major thrust of this work involved the extraction of maintenance knowledge from annotated dataflow designs. One goal was to provide the user with assistance in making changes to the program based on proposed modifications to the design. Via learned dependency information, the system proposed additional changes based on user proposed changes. In their example, based on an accounting system, the user could propose adding a new distribution outlet or deleting a class of products and the system would then identify affected design modules and suggest deletion or addition of design nodes.

The software engineering environment described in this paper through the use of feature contexts also has the goal of maintaining maintenance knowledge concerning a component. The major difference is that through the linking of designs and code, the proposal of new features (such as instantiating a new distribution outlet from a generic feature context) results not in suggestions for how the design might be modified but in actual changes to the code implementing the designs. In this way, a system employing feature contexts has many of the properties of an application generator capable of producing a range of implementations based on variations in the design specifications. It is the feature contexts which encode the maintenance and domain knowledge necessary to construct these application generators. It also means that the system described in this paper is a kind of application generator *generator*, since it is used to construct instances of application generators.

3 Environment Description

The linking of designs and implementations along with the capabilities derived from this linking require the support of an editing environment. This subsection describes the structure of that editing environment and its use in constructing linked designs and programs.

The editing environment will consist of a structure-based editor capable of editing both designs and program code. The use of a structure-based editor allows easy linking of designs and code by representing links as references to the appropriate syntactic structure. In actuality a single logical link such as the link from a design feature to its implementation may involve multiple structure references since the implementation can be distributed throughout the program and therefore not form a contiguous syntactic structure. In addition to supporting the linking information, the environment must also be able to perform the extraction and instantiation functions and monitor dependencies between program and design elements. Extraction requires that the environment maintain information about the relationships between the different features so that interactions between features are handled properly during extraction operations. These relationships can be recorded during the linking process described below and by analyzing data and control flow information from the implementation. This information is encoded as typed links between feature nodes in the design. In order to perform instantiation, the environment must be able to record and play *scripts* associated with generic features. The scripts are constructed when the features

are defined and are stored along with the feature node. Scripts store the information defining the locations in the code at which changes are made during generic feature instantiation. Also, each time a feature is instantiated, an entry is made in the instance list of the generic feature's design node. This allows the user to locate all instances of a feature and also partially automates the updating of existing feature instances in response to changes in the generic template.

The actual linking process involves establishing the correspondences between the design and implementation. There are a couple of very similar ways that this can be done. The two methods are complementary and can be used interchangeably. The first method can be used when creating new software. Once a design has been constructed, an implementation is started. During editing of the new source code, a window containing the portion of the design being implemented is maintained on the screen. Before adding a new section of code, the programmer indicates his work context by clicking on the appropriate design element. All additions until indication of the next design context change apply to the current design element. In the second method of establishing links, both the design and code exist. A link is established again by marking the appropriate design element and then marking one or more sections of code, thus indicating that this code implements the given design element. Note that the methods provide for the description of distributed plans since they allow more than one section of code to be linked to a single design element. It is also possible that several design elements will share a single piece of the implementation. In such a case, various relationships are possible between design contexts. It is possible that design contexts can share pieces of code or that one design context can be embedded within another. These different relationships have implications for how two design contexts interact during extraction and instantiation operations. If sharing is taking place the environment will prompt the user for further information on the type of interaction between the two features.

Constructing non-generic features is simply an extension of the linking process described above. A feature node is constructed documenting the structure of the feature under consideration. Items within this feature node are then linked to the appropriate points in the implementation via the marking process described above. These links and any dependency/sharing information obtained during the linking are all that is required to facilitate the extraction of features.

Generic feature designs consist of two types of elements: static elements and parameter elements. In the construction of generic features, the linking process is used to establish a link between the static elements of a generic feature's design node and program structures which will implement them. Parameter elements are also defined for the generic feature and listed in the design node. No links to parameter implementations are made at this time since these implementations will be supplied at the time of feature instantiation. Constraints and dependencies can be associated with these parameters to aid in their instantiation. Additionally, a script must be defined to show the system where each static and parameterized implementation element is inserted into the existing code. Simple insertion can be handled automatically, however, insertion of some elements may require user intervention and this can be indicated at script creation time. Scripts can be created in much the same fashion as linking. Items in the design node are linked or anchored to locations in the existing code. If no further information is indicated then the insertion can take place automatically at instantiation time. If, however, user intervention is specified, the system will prompt for documentation on what kind of intervention is necessary. This documentation is then displayed to the user at instantiation time.

4 An Example

The example to be presented is based on one found in [4]. It consists of a simple database which is maintained in memory as long as operations on it are being performed. Once the operations are complete, the database is then written to a file on disk for storage purposes. The next time the database is needed, it is read in from the appropriate file and loaded back into RAM.

The database application itself is divided into two subcomponents - a command processor subcomponent and a database subcomponent. The database subcomponent contains all of the database functionality of the application but lacks the interactive interface provided by the command processor. When the user enters a command, the command processor calls an interface method (typically defined in the Interactive_Database component). This method handles all necessary user interaction and parameter parsing and calls an appropriate Database_Engine method to perform the rest of the operation. A specification of the high level design of the database application is given in figure 3. It represents the highest level of a hierarchical object oriented design described in the previous section.

The design presented in figure 3 contains a number of separate sections which serve to organize portions of the design. The first section is simply the name of the component. The name is followed by a short overview of the component which is intended to provide the user with basic information concerning the component. After the overview, the internal state of the component is described. The "::" notation is intended to indicate that the state elements are "inherited" from the indicated subcomponent. In general this is a domain oriented view of the component's state whose implementation may be realized in a number of ways. The linking will make this mapping to realization explicit. The next section is the public interface. The public interface consists of all interface methods which are available to components which include this component. Those methods of the component which are for internal use only are described under the next heading, internal functions. Further decomposition of the structure of these interface functions is performed separately in the design documentation. Following the interface sections, is the list of subcomponents used by the component. In this case, command processor and database engine subcomponents are included and several of their interface methods are inherited by the interactive database application. Finally the features section provides a list of component features which are organized into feature contexts as described above. For the database application, each of the interactive commands is described as an instance of a generic *interactive command* feature. This generic feature will be used shortly to demonstrate the instantiation of generic features. The exception feature, on the other hand, provides an example of the use of the linked structure to localize logically related elements of the design which can be scattered throughout the implementation. In the case of the exception feature, all instances of exceptions in the interactive database application are linked to this feature. From here, each specification of each exception instance can be accessed directly and from each instance, the code that raises and handles each exception can be accessed. Thus all information on exceptions is localized at one point in the design and forms a secondary hierarchy of design information.

In figure 3, all of the bold faced items provide links to associated documentation and/or implementations. This linking allows nodes to be brief and quickly comprehended yet provide details via pointers to other nodes.

An example of one of the local interface methods is presented in figure 4 in a fashion similar to the preceding component. It includes sections giving the name and overview of the method along with

Component Name: **Interactive_Database**

Component Overview: This component is an interactive database which allows a table of records to be read in from a file, manipulated and then written back to disk.

Internal State: **Table of Database_Records::Database_Engine**

Public Interface Functions: **Execute_Command::Command_Processor**

Internal Methods **Read_Database**
Write_Database
Add_Record
Search_Record
Delete_Record
Initialize::Database_Engine
Terminate::Database_Engine

Components Used: **Command_Processor**
Database_Engine

Features: **Interactive_Commands**
Exceptions

Figure 3:

a high level specification of the structure of the function performed. From each line of the structure specification, there is an implementation link to the code that corresponds to that portion of the specification. These links allow the system to help maintain consistency between the documentation and the code. If one is modified the system can highlight the affected area in the other. This means the user can make design changes and then be directed to the affected portions of the implementation. The feature listed in this design node is a sub-feature of the exception feature listed in the Interactive_Database component. The last part of the method specifications is a section called dependencies. This section identifies dependencies which exist between the component containing the dependency and other components. Dependency information is important for insuring proper maintenance of delocalized plans in the design and implementation. In figure 4 a nonexistent file exception is handled which is raised in the Database_Engine method of the same name. Therefore, a dependency exists between the code that handles the nonexistent file exception and that which raises the exception. Any change to either piece of code could require a change in the other. The explicit recording of the dependency allows the system to monitor the code and bring the dependency to the programmer's attention if modifications are made to either piece of code (or to

portions of the specification corresponding to the code). Such notification is an example of the use of active documentation.

Method Name: **Read_Database (Parameters: String)**

Method Overview: This method reads a database file and loads it into the internal table for processing. Non-existant file exceptions are also handled.

Method Structure: **Get the file name string from the parameter string;**
Call Read_Database method in Database_Engine;
Handle nonexistent file exception;

Features: **Exception Handling**

Dependencies: **Exception raised in Read_Database::Database_Engine**

Figure 4:

The structure of the interactive command feature (see figure 5) is similar to the component and method specifications given previously. The instances section lists all instances of features of this type (Interactive_Command) and the Parameters section lists all of the parameters which must be supplied when a new instance of this feature is created. In this case, there are three such parameters. The first, the invocation string, is the command line name used to invoke the new command. The second parameter is the interactive command method. It is the method, native to the interactive database component, used to parse the parameters and call the database engine command. The final parameter is the database engine command to perform the desired operation. Documentation describing the nature of these parameters is attached to each of the parameters. The Invocation_String parameter is merely a string. The other two parameters are methods and require design nodes to document them.

Associated with each generic feature is a script which guides the instantiation of the feature. The script, either directly or in conjunction with the user, makes modifications to both the design documentation and the implementation. This script is constructed by indicating addition and deletion points in the design and implementation. The points of addition are then linked to information sources which will provide the text of the addition. Typical information sources are parameters supplied during instantiation and text fragments which are stored with the script when it is built.

The script for the interactive command feature performs the following operations:

- Insert the **Interactive_Command_Method** as a private method of the **Interactive_Database** component.

Feature Name: **Interactive_Command**

Feature Overview: This feature organizes all interactive commands in the database and provides for addition and removal of commands.

Instances: **Read_Database**
Write_Database
Add_Record
Search_Record
Delete_Record

Parameters: **Invocation_String**
Interactive_Command_Method
Command_Method

Features: **Commands::Database_Engine**
Commands::Command_Processor

Figure 5:

- Insert the **Command_Method** as a public method of the **Database_Engine** component.
- Insert the **Invocation_String** and a call to the **Interactive_Command_Method** into the code that implements the selection of commands in the **Command_Processor**.

The first of the above operations, in addition to modifying the design documentation, adds the implementation of the method to the source code of the system. The same is true of the second operation. The third operation, extracts the name of the Interactive_Command_Method and uses it along with the Invocation_String to modify the code which selects the command to execute. This code may be a simple case statement and the modification may consist of the addition of another selection clause to the case statement where the label is the Invocation_String and the code executed is a call to the Interactive_Command_Method. It is easy to envision such an addition occurring without user intervention. Suppose, however, that the construct implementing the command selection involved a binary search of the command names in a table which must be kept in sorted order. In this case, the script might highlight the table, inform the user of the requirement of maintaining the table in alphabetical order, and request the user supply an insertion point for the label. In either case, significant assistance in making the enhancement is provided to the programmer either by completely automating the changes or by providing guidance as to where and how to make the changes.

Suppose an enhancement to the personal database program is desired which involves the addition of a help function to provide documentation on available interactive commands. This enhancement

will be used to further demonstrate the capabilities of the generic features. The help feature will itself be implemented as an interactive command. The implementation of the help feature will also entail modifications to the Interactive_Command feature of the Interactive_Database. These modifications in turn must be reflected in each of the already instantiated interactive commands.

The design of the help function will be quite simple. It will consist of a set of keys which are the command invocation strings and a set of corresponding documentation strings describing the commands. When the user desires help for a command, the word "help" is entered followed by the invocation string for that command. For example: **>help add_record**. Since the help feature is implemented as an interactive command in this system, there will be two functions needed. One function will parse the command and pass the name of the command for which documentation is required to a lookup function in the Database_Engine. This function will then return the documentation to the interactive function which will then display it to the user. Since it is desired that all interactive commands be documented in the help function it is appropriate that some changes be made to the Interactive_Command feature. These changes consist of adding a new parameter and a couple of dependency relationships. In addition, note that the instances section has been updated to reflect the commands which have been instantiated). The new parameter is the help documentation for the interactive command and the dependency relationships are the following. A requirement that the command invocation string and the help key be the same string and a constraint that whenever the functions implementing an interactive command are changed, the user is prompted to update the help documentation for that command (see figure 6).

Now that the changes have been made to the interactive command feature, it is necessary to update the structure of each interactive command. In particular, the documentation for each command must be supplied. Since the system maintains a list of the instances of the Interactive_Command feature, the system can guide the update process by prompting the user for documentation on each command thus insuring complete implementation of the help command. It can also guide the insertion of the help text as it did in the command selection code of Execute_Command. Such guidance may not seem important but the programmer may not be familiar with all commands or some commands may have a tendency to be overlooked (e.g. the help command itself). It is also easy to envision more complex features with many instances in which the capability to reinstantiate or modify the instantiation is very valuable.

The presentation of the interactive database program in this section has provided an opportunity to demonstrate some of the ideas developed in previous sections. In particular, the enhanced documentation provided by the multiple design views allowed localization of distributed features, recording of delocalized plans and dependency information, and facilitated the use of active documentation to assist in ensuring consistent and complete maintenance of software employing the linked views. The use of generic features to automate or guide the enhancement of software using predefined features contexts was also demonstrated. The documentation capabilities and generic features provided for an evolutionary approach to the development of the interactive database. These same capabilities promote a reuse oriented strategy by allowing quicker understanding and customization of reused components.

The utility of many of the capabilities demonstrated in this section, however, is best motivated by larger, more complex components/applications than the simple database presented here. One capability which becomes increasingly valuable as applications grow more complex is the extraction of features from a component or program. Feature extraction was not demonstrated in the context of the database example. Instead, consider as an example an expert system tool developed at

Feature Name: **Interactive_Command**

Feature Overview: This feature organizes all interactive commands in the database and provides for addition and removal of commands.

Instances: **Read_Database**
Write_Database
Add_Record
Search_Record
Delete_Record
Help

Parameters: **Invocation_String**
Interactive_Command_Method
Command_Method
Command_Documentation

Dependencies: **Equality of invocation string and help key**
Command documentation dependence on implementing methods
Features: **Commands::Database_Engine**
Commands::Command_Processor

Figure 6:

Georgia Tech known as GEST which provides a shell for building rule based systems [5]. Some of the features of GEST include the ability to store knowledge as simple assertions (facts) or in a hierarchical frame tree. Other features provide for an explanation capability, automatic processing of certainty factors, and truth maintenance. Each of the additional capabilities adds processing overhead to the system during inferencing. As a result, users might desire a customized system without some features such as the explanation capability or truth maintenance. Unfortunately, the implementation of each feature is distributed throughout the application. If the expert system shell were constructed with each feature isolated into a separate feature context, then the extraction of any combination of these features could be accomplished with considerable ease. The alternative is a tedious, error prone cycle of searching and modification by hand. Feature extraction is also helpful for a number of other reasons. For example, there may be a need to reimplement some feature of an application in which case the existing implementation of the feature is easily eliminated from the code and the linking associated with the feature provides a guide for the new implementation. In the case of reusable components, it may be desirable to provide several mutually exclusive interfaces to or implementations of a given feature for compatibility or optimization purposes. Feature extraction can then configure the component and allows changes to any of the common code to be made in only one place instead of in each separate version of the component as would be necessary without

the feature extraction capability [6].

5 Related Work

This section will briefly cover several research efforts related to the work discussed in this paper. These works include strategies for linking together multiple representations of a program, systems which allow slicing of programs along various lines, and systems for integrating separate slices into a single unit.

5.1 The Grid Mechanism

The Grid mechanism of Harold Ossher [7] is designed to allow the documentation and description of a hierarchically structured object oriented system from multiple levels of abstraction. This description takes the form of a grid or matrix where the columns represent individual objects and the rows depict these objects from differing points of view. The structure of an object from a particular viewpoint is referred to as a *unit*. One of the primary aims of the grid mechanism is to facilitate the recording and definition of actual and permissible interactions between units of a program. Ossher develops a number of mechanisms for structuring objects and views and for describing interactions between individual and groups of views and objects. The grid can then be used to automatically ensure that a program based on the grid does not violate any of the interaction constraints specified in the grid. Features, on the other hand, are intended to provide structures which guide and control configuration and modification of program components. The designs used in the feature-based approach also concentrate on the documentation of program function as well as component interaction. Additionally, units in the grid mechanism generally correspond to collections of procedure level objects so that the grid mechanism does not deal with the description of the fine grained structures which are present in features.

5.2 Hypertext Systems

At the core of hypertext its ability to link and organize related pieces of information. These capabilities along with the introduction of high performance interactive workstations has prompted the use of hypertext in a number of information/database systems. In particular, it has been applied to the area of CASE tools for the purpose of effectively documenting program structure. The most prominent system of this type is Neptune [8]. Neptune is a hypertext CAD database which has been targeted to software engineering. Neptune uses a directed graph of nodes to model the structure of program and documentation text. Attributes attached to nodes describe the content of nodes. Links and attributes attached to them allow documentation of arbitrary relationships between nodes. As has been the case with other CASE databases [9] [10], Neptune concentrates on documenting the coarse structure of the software (ie at the level of modules and procedures). This research differs from that of Neptune and the others in both the detail of the correspondences established and in the structures and operations under consideration. The definition and use of feature contexts provides capabilities for configuring and understanding software not envisioned by Neptune. Some of the configuration capabilities of feature contexts are captured in the PIE system [11]. This system allows for the possibility of automatically generating alternate implementations

of smalltalk applications from stored configuration information. However, the stored configuration information is more coarse grained than is possible with feature contexts and no provision for generic instantiation is made.

5.3 Delocalized Plan Documentation

Research into the area of delocalized plans [4] is also closely related to the concept of feature contexts. Plans, which are similar to cliches in the Programmers Apprentice project [12], are abstract computational models for performing a particular task in a program. A key feature of delocalized plans is that the code that implements them is not confined to one location in the program. A couple of solutions to the recognition of delocalized plans have been proposed in [4]. The first solution is based on the use of fairly standard documentation to explicitly reference the delocalized plans. Tests of this form of documentation met with some success but were not universally successful in alerting programmers to the existance of the plans. The second solution involves a more active approach. An integrated documentation/program editing environment is used to monitor the program variables being modified by a programmer. Since delocalized plans are almost always closely associated with a collection of data elements, the monitoring allows the system to detect when a programmer is modifying code associated with a distributed plan and alert the programmer to the plan's existence. No details regarding the effectiveness of this solution are currently available.

The environment described in this paper can also document delocalized plans by isolating all of the implementation components of the plan (both data and code) into a feature context. Information on the structure and role of the various elements of the plan can then be linked to the implementation. Feature contexts, as described in this paper however, are not limited to the documentation of delocalized plans. A number of interesting program constructs do not fit the definition of a plan. Such features as error checking code and the changes which need to be made when a new "shape" is added to a drawing program provide examples of non-plan constructs which can be captured by features. The concept of a program view or slice as discussed in this paper and a number of others [7] [13] is better able to document these more general features. Feature contexts go a step further, than simply documenting these features, by allowing the manipulation of these features via extraction and instantiation. This ability to manipulate the program code from the level of the design is a significant advance over plan and other view based systems.

5.4 Views of Tools

The approach taken by David Garlan in his thesis [14] and reflected in [15] and [16] provides another use for views of programs. The general concept of a view of a program as a slice or perspective is encountered in a number of works discussed in this section. Garlan's treatment of views, however, introduces the idea of using views in the synthesis as opposed to the documentation of programs. Through a process called merging, which is related to the concept of multiple inheritance found in some object-oriented languages, a single object definition is created from several view definitions. The merging process provides for automatically maintaining the consistency of data items defined in separate views which represent the same data entity but which may have different data types. Another example of this form of views, referred to as *logical views*, is found in [17]. The logical views approach requires that all data shared between two views be of the same type but allows

multiple instances of a given view type within a single object. These multiple instances allow a *temporal*[2] sharing of data that is not possible in the preceeding approach.

The approach presented in this paper, also uses view (features contexts) as a basis for synthesizing code The differences lie in the integration of design information with code to construct feature contexts. In the environments described above, views are a partitioning of implementation level constructs and make no use of design information. The design information in feature contexts allows for multiple decompositions of an implementation along different conceptual lines. Views, as defined in Garlan's thesis and in the logical views approach, also involve only sharing at the level of whole data items or procedures. Thus, as a tool for configuring software, views suffer from the *same as except* problem just as object oriented languages do. Feature contexts allow sharing of any syntactic entity so features can be defined without restrictions based on the implementation language.

6 Conclusion

It has been argued that object-oriented techniques provide a solution to the *same as except* problem and to an extent this is true. Through inheritance and overloading, new classes can be defined which are the same as an existing (ancestor) class except for some new or redefined methods or data items. However, object-oriented programming languages are limited in that the variations provided by these mechanisms must still be made at the level of whole methods and data items. Often times the necessary changes cannot be made by simply substituting one or two new methods or variables. This situation arises, for example, when a system built from reusable components is being optimized. Initially the components, being robustly designed, contain a considerable amount of error checking code. If at runtime this checking results in unacceptable overhead, it may be necessary to eliminate the error checking. The problem arises because, in all likelihood, there are not one or two error checking methods which can be redefined or removed but a sprinkling of error checking code throughout all the methods. In this situation object-oriented techniques are of no help. On the other hand, with proper feature definition, (ie a feature organizing all error checking) the extraction of the error checking code is trivial.

A software engineering environment as described here directly addresses the modifiability problem by explicitly linking software designs to implementations - localizing and partitioning the portions of the implementation corresponding to distinct features and thus facilitating understanding. Feature extraction and generic feature instantiation also provide mechanisms for partially automating the configuration of the software. Furthermore, in the context of software reuse component modification is simplified, thus promoting construction of software via reusable components. Software constructed from such components can be assembled quickly and inherits the reliability and maintainability of the reusable components from which it is built.

References

[1] Grady Booche *Software Components with Ada* Benjamin/Cummings Publishing Co. Menlo Park, CA 1987

[2] Temporal sharing involves data that is sometimes shared between views and sometimes is not.

[2] S. Horwitz, T. Reps, D. Binkly "Interprocedural Slicing Using Dependence Graphs" *Procedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation* June 22-24, 1988 Atlanta, GA; ACM Press

[3] Vasant Dhar, Matthias Jarke "Dependency Directed Reasoning and Learning in Systems Maintenance Support" *IEEE Transactions on Software Engineering* Vol. 14, No. 2, February 1988

[4] Elliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, Robin Lampert "Designing Documentation to Compensate for Delocalized Plans" *Communications of the ACM* Vol. 31 No. 11, November 1988; pp. 1259-1267

[5] Stephen D. Tynor, Stefan P. Roth, John F. Gilmore "GEST: The Anatomy of a Blackboard Expert System Tool" *Proceedings of the 7th International Workshop on Expert Systems and Their Applications* Avignon France, May 13-15 1987 Vol 1 pp. 793-808

[6] Franz J. Polster "Reuse of Software Through Generation of Partial Systems" *IEEE Transactions on Software Engineering* Vol. 12 No. 3 March 1986, pp. 402-416.

[7] Harold L. Ossher *A New Program Structuring Mechanism Based on Layered Graphs* Ph.D. Thesis, Stanford University, December 1984, pp 236.

[8] Norman Delisle, Mayer Schwartz "Neptune: a Hypertext System for CAD Applications" *Proceedings of ACM SIGMOD International Conference on Management of Data* Washington, D.C., May 1986 pp. 132-143

[9] Maria H. Pendo, E. Don Stuckle "PMDB - A Project Master Database for Software Engineering Environments" *IEEE Proceedings of the 8th International Conference on Software Engineering* August 1985, pp 150 - 157

[10] Bertrand Meyer "The Software Knowledge Base" *IEEE Proceedings of the 8th International Conference on Software Engineering* August 1985, pp 158 - 165

[11] Ira Goldstein, Daniel Bobrow "An Experimental Description-Based Programming Environment: Four Reports" Xerox Corp. CSL-81-3 March, 1981

[12] Charles Rich, Richard C. Waters "The Programmer's Apprentice: A Research Overview" *IEEE Computer* Vol. 21 No. 11 November 1988, pp. 10-25

[13] Mark A. Linton "Implementing Relational Views of Programs" *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* pp 132 - 140 Pittsburgh, PA, 1984

[14] David Garlan *Views for Tools in Integrated Environments* Ph.D. Thesis, Carnegie Mellon University CMU-CS-87-147, 1987, pp 143

[15] A. N. Habermann, Charles Krueger, Benjamin Pierce, Barbara Staudt, and John Wenn *Programming with Views* Technical Report, Carnegie Mellon University CMU-CS-87-177, 1987, pp 139

[16] Gail Kaiser and David Garlan "Melding Software Systems from Reusable Building Blocks" *IEEE Software* July 1987, pp 17 - 24

[17] John J. Shilling, Peter F. Sweeney "Three Steps to Views: Extending the Object-Oriented Paradigm" Proceedings of OOPSLA'89, New Orleans, LA, 1989

Evaluating Software Language Technologies

Kenneth Dickey
Tektronix, Inc.
PO Box 4600, MS 92-101
Beaverton, OR 97076
kend@mrloog.LA.TEK.COM

Abstract: There are significant differences in software technologies. Newer compiler technology and code transformation techniques are changing cost equations. Tools and methods appropriate for developing software for device drivers and small (under 50K LOC) EPROMed projects fail miserably when applied to large open systems and vice versa. Concepts easy to express in one programming language may be nearly impossible in another. Project budget, goals and constraints define a shape that a software technology must fit into. This talk will focus on metrics, issues, and tradeoffs in choosing an appropriate software technology for your next project.

Bibliography: Kenneth Dickey received his Masters Degree in Computer Science from Purdue University in 1982. After graduation, he participated in the design and implementation of a commercial programming environment running on a multiprocessor mainframe. For the last 4 years he has worked for Tektronix, Inc. Recently, Ken participated in the development of a Scheme compiler for the Motorola 88100 RISC Processor. He is currently the project software architect for a development group in Tektronix's Logic Analyzer division.

Evaluating Software Language Technologies

Kenneth Dickey
Tektronix, inc.

0. INTRODUCTION

In the building of software systems, there are many areas where traditional development methods are failing. An all too frequent symptom is having development take twice as long as is scheduled. Time to market is critical to picking up market share. Changes in software technology are making possible the rapid building of high quality software systems. In order to make use of newer technologies, we must rethink our software development strategies.

I believe a major problem is that we are trying to build software systems using technologies appropriate for device drivers and small systems, and the technologies don't scale up. Complexity has overtaken execution speed as the largest dragon to be faced. Effectively managing this complexity in the context of market driven software development makes certain demands on software language technologies. This shift in viewpoint in turn influences the criteria by which these technologies are to be evaluated.

One of the reasons usually given for using old technology in development projects (Note that Unix with C is circa 1972)** is that using new technology carries risk. This argument breaks down, however, in areas where traditional technology is not working. In such areas, the use of newer software technologies can reduce risk and have a major positive impact on market share.

The remainder of this paper is structured as follows. Section 1 discusses software development goals. Section 2 sets forth a framework for reaching these goals. Section 3 shows how changes in software technology support this framework. This sets the stage for the evaluation criteria which are presented in section 4. Also included are some illustrative sample points (section 5) and a technical appendix.

1. SOFTWARE SYSTEM GOALS

The overall goal of a software system is sustained sales. The economics of software determines the basis of the engineering goals given below.

1.1 COUNTING THE COST

As software development becomes more market driven, we must base investment decisions on economic criteria. Software systems are very expensive to build. We need to amortise the capital investment in development cost by reuse. To protect this investment requires that software be highly flexible and portable. This increases market share because of the ability to economically specialize products for niche markets.

'Bugs' have been tolerated in software in order to ship systems faster. A basic problem with this strategy is that the cost of managing and fixing these bugs grows rapidly over time. Engineers fixing bugs are not busy building new products. Zero defect software is a goal with an economic basis.

** [Unix is a registered trademark of AT&T Bell Laboratories]

Because of the interactions between zero defects, reuse, and flexibility, these cost/benefits are hard to quantify. One cost which is easier to calculate is time to market which is measured in market share in dollars per month of lost sales. Because it is easier to estimate, time to market frequently overshadows other costs in decision making. Good economics, however, dictates that total costs be considered.

1.2 ENGINEERING GOALS

Given the economic context, engineering goals for software systems can be summarized as follows:

[1] fast construction
[2] zero defects
[3] high flexiblity, portability, and robustness
[4] functionality

There is a definite tension between building software fast and the other goals. Achieving all four goals requires building the proper framework. What kind of a framework is required? How are advances in software technology making this possible?

2. WHAT KIND OF A FRAMEWORK IS NEEDED TO ACHIEVE THE GOALS?

There is a change of emphasis required to satisfy the goals listed above. The traditional approach takes a text editor, compiler, assembler, and linker as fundamental and adds documentation and management. For changeable, reusable software, the emphasis must be on communication of design. What is fundamental is the ability to capture, browse, and update design abstractions. Implementation is subsumed by design.

This can be seen in engineering disciplines in other industries. In general, effective design engineers:

[1] use a consistent design language and notation backed by a formal theory which ties together design, implementation, and testing and is used to communicate design between groups
[2] have knowledge of a variety of implementation strategies and techniques
[3] solve problems in the context of a system and at the appropriate level
[4] are supported by a design environment which implements [1] and [2]

The crux of software systems development is having a technology which gives rapid turnaround in testing design ideas, and has a strategy for building software systems by composition and specialization of designs *[Shaw]*.

3. WHAT IS CHANGING IN SOFTWARE TECHNOLOGY TO SUPPORT DESIGN?

There have been refinements and developments in specification technology, programming language design, and programming language implementation which have been reducing the costs traditionally associated with use of formal design methods and higher level languages. Design and communication is supported by use of formal models. Building abstract interfaces is supported by clean syntax, multi-paradigm support, and the ability to compose and specialize higher order procedures. Rapid development is supported by automatic storage management and the ability to run both compiled and interpreted code in the same environment.

3.1 RAPID PROTOTYPING vs INCREMENTAL DESIGN

In developing software systems, there are two kinds of software: the kind we know enough to specify up front and the kind we don't. Software systems in the large usually fall into the second category and require rapid prototyping support in order to be able to get fast feedback on concrete design decisions.

The main idea behind rapid prototyping is to be able to make mistakes fast and fix them fast. This is particularly helpful for architectural prototyping. Given historic practices, coding starts only after analysis, top-level design and detailed design. Thus, it can be on the order of years for enough of an implementation to be running to give designers feedback on their initial design decisions, by which time market windows are closing and the options are few. Late feedback translates into high risk. Prototyping gives early feedback on feasibility, costs and performance.

Closely related to rapid prototyping is incremental design. In software, it is frequently necessary to interleave design and implementation due to the complexity and long duration of the system being developed. What differentiates rapid prototyping is that "efficiency and completeness are often sacrificed in the interests of rapid development and ease of obtaining information" *[Gabrial89]*. In practice, the differences may be moot *[Lugi]*.

3.2 SPECIFICATION TECHNOLOGY

Specification technology is maturing. By that, I mean that specifications are now being used in industry to solve real design problems. The key idea behind specification is clarity--having a formal model of what is to be done. Where exhaustive testing is expensive and correctness essential, formal proofs may be cost effective.

Implementations may be derived from formal specification *[Barrett]*:

(1) Do the formal specification.
(2) Decompose it into smaller specifications; show recomposition is valid.
(3) Derive programs from the specifications.
(4) Use correctness preserving transformations to make the programs efficient.

3.3 PROGRAMMING LANGUAGE TECHNOLOGY

Poor choice of programming language can cripple an otherwise good design environment.

There are some fundamental qualities of good programming language technologies. The syntax should be amenable to processing by both machines and humans. The semantics should be cleanly specified and unambiguous so that it can be formally shown that different implementations define the same language. A language should support a wide variety of programming styles/paradigms so that solutions can be structured appropriately to problems. The language support should be appropriate for both production and rapid prototyping. A programming language should supply a foreign language interface to allow solutions to be bolted together. Very importantly, development systems and languages should be well founded in theory. The Programming Language, Libraries, and Environment should be well specified and cleanly separable. Typically, the language should be at a high level, supporting abstraction well and hiding details of particular machine architectures. The implementation of the language should be reasonably efficient.

3.3.1 ABSTRACTION: FIRST CLASS PROCEDURES

Crucial to code reuse is the ability to separate out the reusable, generic code from system- and environment-specific code. Such separation is prerequisite to the ability to compose and specialize the generic code to be computationally efficient for a given environment. An effective language implementation technology which gives these capabilities is realized under the heading of first class procedures. First class procedures support both object oriented and functional paradigms *[AdamsRees, AbelSuss85]*. First class procedures support both accuracy and understandability *[HalSus89, Roylance89]*.

For constructs to be first class in a language means to be able to create them without naming them, store them in data structures, pass them as arguments, and return them as results. Numbers are first class citizens in most programming languages--imagine having to name all numbers before use!

First class procedures can be used as parameters to specialize functions which can then be used to specialize other procedures. For example, a numerical integration procedure can be specialized by taking a function to be integrated and returning a function which takes limits of integration and does the actual integral computation. Such a specialized integrator can then be used in several places in a larger numerical model. In the object oriented paradigm, such parameterization takes place via the mechanism of inheritance. Languages such as C and Pascal do not allow unnamed procedures or functions and do not allow functions to be specialized and then returned as results. These restrictions severly limit the expressiveness of such languages by disallowing the composition of higher order procedures.

3.3.2 EXPRESSIVENESS: SUPPORT FOR MULTIPLE PARADIGMS

The more programming paradigms (models) a language supports, the greater its ability to structure appropriate solutions. Here is a thumbnail discussion of key ideas behind some major programming models. [Comments on usefulness should be taken as illustrative only].

Imperative programming means programming with assignment statements. It is typical in such languages as C and Pascal. One may think of operating on data at particular machine addresses.

Object oriented programming is typically a structured form of imperative programming in which data is encapsulated and access routines are used to change data in ways which (hopefully) guarantee its consistency. It is useful in user interfaces and for simulations of physical objects where structured data relationships are the focus of attention.

Functional programming is based on functional composition and in its pure form disallows assignment. It is useful in numerical applications and process based computation where simple dataflow (i.e. "streams") or interesting and complex control flow is the focus of attention (see also *[Jones]*). Functional programming is sometimes called **applicative** programming.

Declarative programming uses relations (constraints) or numerical and/or logical equations which are used by some underlying system to derive solutions or solution systems. Equational, Constraint Based, and Logic programming are seen as declarative. Declarative programming is useful in general problem solving but is frequently inappropriate where execution speed is at a premium. There is much work going on in this area to improve computational performance.

Access oriented programming uses the idea of 'active values'. When the value of a variable changes, some action is taken (e.g. update a screen image of a meter). This can be done by the runtime system without changing the algorithmic source code. This is useful in graphical user interfaces.

Transformational programming is a term used to describe a process of taking a correct but possibly inefficient program and, using standard transformations, change it into a more efficient program. A key idea here is that the optimized program is typically much more complex and difficult to understand and prove correct. When a change in requirements is made, the change is made to the simpler original program which is then transformed again to a more efficient form. This process is not yet fully automated, but semi-automated systems are in use today. Knowledge of transformational techniques is useful in general, even without automated support.

It is important to note that the above paradigms are not mutually exclusive. For example, functional style control abstractions may operate on object oriented data.

The most important thing to be aware of is when a particular paradigm is inappropriate to the solution you are trying to achieve. It has been suggested by some, for example, that checking spelling as one is typing is not appropriate because it does not fit into the pipe model of Unix. Another way of viewing this is that the pipe model is not a good fit for implementing an incremental spelling checker.

Languages with high paradigmicity are not necessarily more complex than others. Scheme *[ReesCling86]* is an example of such a language. It has a small set of features which can be composed without restriction and in addition is highly expressive and computationally efficient (see code sample in the technical appendix).

3.3.3 MIXED CODE

Rapid development requires fast feedback. The ability to interpret code for fast feedback and good debugging and then compile code for speed gives the best of both worlds. The ability to mix interpreted and compiled code in the same environment means that a small code change can be made and tested without having to wait for long compile, link, load cycles. Mixed code also supports incremental development. Efficient support for mixed code usually requires automatic storage management.

3.3.4 AUTOMATIC STORAGE MANAGEMENT

Automatic Storage Management, sometimes referred to as Garbage Collection, can reduce project development time by 1/3 to 1/2. The reason that this is so has to do with the mistakes made and time consumed with storage management and explicitly dereferencing pointers, as well as development cycle time. This was uninteresting when automatic storage management used 40% of the available CPU cycles. However, today's algorithms use less than 2% of the CPU *[Ungar, Appel]*. Compilers for newer languages such as Scheme and ML which use automatic storage management are providing runtime performance competitive with C and Pascal while supporting multiple paradigms and rapid development *[e.g. Rees84, Kranz]*. automatic storage management is not free--there are space costs and care must be taken in real-time code, but given that shortened development time translates into larger market share, the question has to turn from "Why should I use automatic storage management?" to "How can I afford to use a system which does not provide automatic storage management?".

3.4 STRATEGY REVISITED: STRATIFIED DESIGN

Software is frequently hard to change. Many seemingly small changes in behavior require an amount of work to implement which seems totally out of proportion with the magnitude of the changes requested. To make such changes more tractable, system complexity must be managed in a certain way. Ableson and Sussman have established guidelines to this end *[AbelSuss86]*:

(1) Create design abstractions to hide detail and separate specification from implementation.
(2) Describe a design in terms of levels of description -- at each level small design modifications should require only small changes in description.
(3) Establish conventional interfaces which allow the combination of components in a mix and match way.
(4) Make languages to describe designs in appropriate ways -- emphasize some aspects and suppress others.

By stratifiying design into layers--each layer having a set of primitives which can be combined in a stylized way--design inscriptions more readily match design abstractions. This strategy is more general thanthat presented in section 3.2. It facilitates change and emphasizes the connection between design and implementation. Ideally, the gap between the programming language notation and the design notation approaches zero *[AbelSuss88]*.

4. EVALUATING SOFTWARE LANGUAGE TECHNOLOGIES

Implementations should be judged on how well they support and leverage design expertise to carry out engineering goals.

4.1 ZERO DEFECT DESIGN

The ability to accurately and precisely capture designs is crucial to zero defect design. To capture and communicate formal models requires an expressive specification language. Automated support for such a language reduces errors and increases effectiveness of its use.

Realization of design in a programming language is enhanced by the ability to do stratified design. Key areas to look for are:

[1] understandability: clear syntax and semantics
[2] abstraction support -- notably 1st class procedures
[3] amenability to program transformations
[4] expressiveness -- support for multiple paradigms
[5] composition methodology
[6] the ability to relate code to design

4.2 RAPID DEVELOPMENT

Rapid development is founded on code and design reuse and rapid design realization. An implementation should be as close as practical to a full rapid prototyping environment *[see Gabrial89]*. Code reuse implies good mechanisms for specialization and composition. Rapid turnaround is supported by automatic storage management and the ability to run mixed compiled and interpreted code. The ability to reuse and coexist with older designs requires a foreign language interface.

It is important in this area to differentiate development and delivery environments. An important aspect of rapid development is scalable technology. For example, automatic storage management is important for rapid development, but a delivered product may not require it or may need a different implementation than is used in the development environment. Compiled code for a delivery system typically has different characteristics from that required for rapid development and analysis. Analysis tools should be available in the development environment to size and minimize system resource use in the delivery environment. It is very important given such differences in environment that the language semantics do not change. Software implementation technologies which support multiple strategies, change, and scalability are to be preferred.

4.3 EFFICIENCY

Efficiency measures for language implementation are fairly well understood and will not be covered here. Language implementation efficiency is important. A language implementation should provide analysis tools to make production code as efficient as it needs to be. What is crucial, however, is design efficiency as measured in the ability to rapidly bring a series of products to market.

4.4 MATURITY

Language technologies of sufficient breadth to handle current software engineering problems are not built by a few people. A community effort is required. Being part of a community means not having to pay the cost of doing everything oneself. An interesting language technology has a research community injecting new ideas. A mature language technology has a user community to beat ideas into shape. Measures of language maturity are the quality of ideas expressed as formal models in academic papers, the quality of their concrete implementations, availability of educational materials, and usage in significant applications.

5. SAMPLER: Unix, C, C++, Smalltalk, Z, Scheme, CIP

In order to make some of the above discussion more concrete, a few sample points will be used for illustration.

5.1 THE UNIX TOOLSET

The Unix interface and toolset has been successful as a traditional programming environment. There are, however, some significant problems with a 'vanilla' Unix environment when we try to reach the goals outlined above. The tools are ad hoc, have different interfaces, and do not support structure above the bytestream level. For example, it is not uncommon to grep a file looking for a pattern, transliterate it, and use awk and/or sed to edit the result which then may be passed again to other tools. But grep, tr, sed, awk, and the shells all use different command syntax! Structure information is repeatedly built and thrown away without being shared among tools. Discovering information about the system is difficult and most documentation so poor that Unix becomes an "insider" system which is not very usable until after much study and experimentation. The poor documentation and baroque interfaces make training expensive, so insiders (people who have paid their price of initiation) are preferred in hiring and the system perpetuates itself. Given the decreasing cost of workstations and the high cost of training software engineers, the typical Unix interface, although inexpensive, may be a poor investment and is certainly not a good design environment.

Design Environments are too large a subject to be addressed further here. For an example of a fairly fully developed environment see *[Cedar]*. Note also *[SWDProceedings]*.

5.2 THE 'C' LANGUAGE *[HarbSteel]*

C's strengths--its simple computational model, flexible minimalist philosophy--and the fact that C has been tied to the Unix implementation have made it a successful programmiog language. As a semi-portable, high-level assembler, C is used very effectively. Of course, many deficiences in C are also well known *[PohlEdelson]*.

The chief disadvantage to C when used as a high-level language is its lack of abstraction. Because of the pervasive use of "include files", there is no way to separate interface from implementation; code for storage representations and values must be directly imported to be useful. In addition, C does not support the creation or composition of higher-order functions. Finally, C is fairly imperative. It supports multiple styles only to the extent that you build them yourself--just as you can build nearly anything in assembler--but coherent support is lacking.

5.3 THE 'C++' LANGUAGE *[Stroustrap]*

C++ is a good illustration of ad hoc development unsupported by an underlying formal theory. There is no clear separation of the implementation and the language. The semantics are still in flux. The syntax is baroque and complex. Although retaining much of C syntax, C++ is a separate language and will not compile C programs of any significance. C's simple computational model is lost. Although supporting object oriented code sharing, C++ does not support higher-order functions. The language is overly complex. There is no complete language definition.

5.4 THE Smalltalk-80 LANGUAGE AND ENVIRONMENT *[GoldRob]*

Smalltalk supports prototyping and simulation well and has a fairly reasonable development environment for individuals. The lack of separation between the language and the programming environment, however, creates development problems. It is difficult to do group development and source code control because code sharing is done by 'merging' code into runtime images rather than by linking. There is no clean way to separate out the development environment to create a product.

5.5 THE 'Z' (Zed) SPECIFICATION LANGUAGE *[Spivey]*

This specification language is becoming more widely used in industry *[Barret, DesGar, Hayes, LonMil]*. As an example, Z has been used to advantage in the floating point specification used to build the Inmos Transputer. The significant thing about the initial Transputer project is that it was done by a small group in a short time. One team member was a mathematician whose primary task was boolean proofs. The implementation of number systems is a good match for formal specification because of the deep and rigorous understanding needed and the expense and impossibility of exhaustive testing.

5.6 THE Scheme LANGUAGE *[ReesCling86]*

Scheme is an example of good language design. The few language features are regular and can be composed without restriction. Scheme supports a wide variety of programming styles *[AbelSuss85]*. The language is well founded in the theory of the lambda calculus *[Barendregt]*. Its specification via denotational semantics *[CliFreWa, Scott, Schmidt, Stoy]* makes it possible to formally prove that an implementation is correct *[Clinger84]* and has had the effect of making such implementations very reliable. Efficient implementations exist [e.g. Chez Scheme, T, MacScheme] on a wide variety of hardware architectures.

Scheme enjoys a large research and user community. It originated at MIT (circa 1975) and is now taught as a first programming language in over 100 U.S. universities.

5.7 PROJECT CIP--A COMPUTER-AIDED TRANSFORMATION SYSTEM

CIP is a long running project (started circa 1974) of the Technical University of Munich whose aim has been to develop a "wide spectrum language" and transformation system to support the correct transformation of specifications down through low-level code generation *[BauerEtAl85,87]*. By wide spectrum language is meant the support of many different styles (functional, imperative, logic) within a single strongly typed algebraic framework. They have a working prototype and have done an impressive amount of theoretical work. This technology looks very interesting and will probably be here in about the same time frame as C++ (3-5 years).

6. CONCLUSIONS

Programming language technologies should be evaluated on how well they support engineering goals *[section 1.2]*. Engineering goals are founded on gaining competitive economic advantage *[section 1.1]*. Satisfying these goals requires rethinking development strategies in an environment where design is fundamental *[section 2]*. Communicating design requires formal models *[section 3.2]*. This in turn emphasizes aspects of programming languages which support such design strategies in the context of rapid software development *[section 3.3]* and provides guidelines for their evaluation *[section 4]*.

In the early stages of a technology, practice leads theory and ad hoc methods are appropriate. As a technology matures, theory leads practice. Software technologies are maturing to a point where formal methods are useful. The complexity of software systems is mandating their use. This paper has attempted to provide an orientation for evaluating software technologies which takes cognizance of formal descriptive methods and is useful in practice.

7. REFERENCES

[AbelSuss85] {Scheme-usage}
Abelson & Sussman: "Structure and Interpretation of Computer Programs" MIT Press, 1985.

[AbelSuss86] {Scheme-usage}
H. Abelson & G. Sussman: "Computation, An Introduction to Engineering Design", MIT Technical report, 1986.

[AbelSuss88] {Scheme-usage}
H. Abelson & G. Sussman: "Lisp: A Language for Stratified Design", Byte Magazine (McGrawHill), February 1988.

[AdamsRees] {Scheme-implementation}
N. Adams & J. Rees: "Object-Oriented Programming in Scheme", Proceedings of the 1988 ACM Conference on Lisp and Functional Programming.

[Appel] {Automatic Storage Management}
Appel, Andrew: "Garbage Collection Can Be Faster Than Stack Allocation", Information Processing Letters 25 (North Holland), 1987.

[AppelJim] {CPS}
A. Appel & T. Jim: "Continuation-Passing, Closure-Passing Style", Proceedings 16th Annual Symposium on Principles Of Programming Languages (ACM), January 1989.

[Barendregt] {Lambda Calculus}
H. Barendregt: "The Lambda Calculus: Its Syntax and Semantics", Studies in logic and the foundations of mathematics, Volume 103, North-Holland 1984.

[Barrett] {Z}
G. Barrett: "Formal Methods Applied to a Floating-Point Number System", IEEE Transactions on Software Engineering Vol 15, #5, May 1989.

[BauerEtAl85] {CIP}
F. Bauer et al, The Munich Project CIP: Volume I: The Wide Spectrum Language CIP-L", Lecture Notes in Computer Science, Vol 183, Springer-Verlag, 1985.

[BauerEtAl87] {CIP}
F. Bauer et al, The Munich Project CIP: Volume II: The Program Transformation System CIP-S", Lecture Notes in Computer Science, Vol 292, Springer-Verlag, 1987.

[Cedar] {Cedar}
D. Swinehart et al: "A Structural View of the Cedar Programming Environment", ACM Transactions on Programming Languages and Systems, Vol 8 #4, October 1986.

[Clinger84] {Scheme-compiler: proof of correctness}
W. Clinger:"The Scheme 311 Compiler, An Exercise in Denotational Semantics", 1984 Symposium on Lisp and Functional Programming.

[Clinger87] {continuations}
W. Clinger: "The Scheme Environment: Continuations" Lisp Pointers, Vol 1 #2, June-July 1987.

[ClFreWa] {denotational semantics}
Clinger, Friedman & Wand: "A scheme for a higher-level semantic algebra" in "Algebraic methods in Semantics", Ed: Nivat & Reynolds Cambridge U Press, 1985.

[DesGar] {Z}
N. Deslile & D. Garlan: "Formally Specifying Electronic Instruments", Proceedings of the 5th International Workshop on Software Specification and Design (also ACM Sigsoft Engineering Notes, Vol 14, #3, May 1989).

[Dybvig] {Scheme-language}
K. Dybvig: "The Scheme Programming Language", Prentice-Hall, 1987.

[FrlHaKobl84] {continuations}
Friedman, Haynes, & Kohlbecker: "Programming with Continuations", in "Program Transformations and Programming Environments" Ed: P. Pepper, Springer-Verlag 1984.

[FrlHa85] {continuations}
Friedman, Haynes: "Constraining Control". Proceedings 12th Annual Symposium on Principles Of Programming Languages (ACM), January 1985.

[Gabrial89] {Rapid Prototyping-requirements}
R. Gabrial: "Draft Report on Requirements for a Common Prototyping System", SIGPLAN Notices, Vol 24 #3, March 1989.

[GoldRob] {Smalltalk-80}
A. Goldberg & D. Robson: "Smalltalk-80: The language and its implementation", Addison-Wesley, 1983.

[Hayes] {Z}
I. Hayes (ed): "Specification Case Studies", Prentice-Hall, 1987.

[HaFri87] {continuations}
Haynes & Friedman: "Abstracting Timed Preemption with Engines", Computer Languages, Vol 20 #2, 1987 (pub Great Britain).

[HaFriWa84] {continuations}
Haynes, Friedman, & Wand: "Continuations & Coroutines", 1984 Symposium on Lisp and Functional Programming.

[HalSus89] {Scheme-usage}
M. Halfant & G. Sussman: "Abstraction in Numerical Methods", Proceedings of the 1988 ACM Conference on Lisp and Functional Programming.

[HarbSteel] {C}
S. Harbison & G. Steel: "C: a reference manual", Prentice-Hall, 1984.

[Jones]
S. Peyton Jones: "Functional programming languages as a software engineering tool", Lecture Notes in Computer Science, Vol 287, Springer-Verlag, 1987.

[Kranz] {Scheme-implementation}
Kranz et al: "Orbit, an Optimizing Compiler for Scheme", SigPlan Notices, Vol 21 #7, July 1986.

[LonMil] {Z}
R. London & K. Milsted: "Specifying Reusable Components using Z: Realistic Sets and Dictionaries", Proceedings of the 5th International Workshop on Software Specification and Design (ACM Sigsoft Engineering Notes, Vol 14 #3, May 1989).

[Lugi]
Lugi: "Software Evolution Through Rapid Prototyping", IEEE Computer, Vol22 #5, May 1989.

[PohlEdelson]
I. Pohl & D. Edelson: "A to Z: C Language Shortcomings", Computer Language, Vol 13 #2, 1988 [Pergamon Press, Great Britain].

[Reese84] {Scheme, T}
Reese, Adams, & Meehan: "The T Manual", 4th Edition, Yale U. January, 1984.

[ReesClIng86] {Scheme-language}
J. Reese & W. Clinger (eds): "Revised^3 Report on the Algorithmic Language Scheme", SigPlan Notices, Vol 21 #12, December 1986.

[Roylance89] {Scheme-usage}
G. Roylance: "Expressing Mathematical Subroutines Constructively", Proceedings of the 1988 ACM Conference on Lisp and Functional Programming.

[Schmidt] {denotational semantics}
Schmidt, David: "Denotational Semantics: A Methodology for Language Development", Allyn & Bacon, 1986.

[Scott] {denotational semantics}
Scott, Dana: "Domains for Denotational Semantics" ICALP '82, Aarhus, Denmark, July 1982.

[Shaw]
M. Shaw: "Larger-Scale Systems Require Higher-Level Abstractions", Proceedings of the 5th International Workshop on Software Specification and Design (also ACM Sigsoft Engineering Notes, Vol 14, #3, May 1989).

[Spivey] {Z}
Spivey, J.M.: "The Z Notation, A Reference Manual", Prentice Hall, 1989.

[Stoy] {denotational semantics}
Stoy, Joseph: "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory", MIT Press, 1977.

[Stroustrap] {C++}
B. Stroustrap, "The C++ Programming Language", Addison-Wesley, 1987.

[SWDProceedings]
"Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments", SIGPLAN Notices, ||Vol 19 #5, May 1984 || Vol 22 #1, January 1987 || Vol 24 #2, February 1989.

[Ungar] {Automatic Storage Management}
D. Ungar: "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm" ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference, April 1984.

[Wand80a] {continuations}
Wand, Mitchell: "Continuation-Based Program Transformation Strategies", JACM V 27, #1, January 1980.

[Wand80b] {continuations}
Wand: "Continuation-Based Multiprocessing", Proceedings of the 1980 Lisp Conference.

8. TECHNICAL APPENDIX: Scheme Samples

It is difficult to give the flavor of what programming in a highly paradigmic language is like in a small example. Here the Scheme programming language is used to give a small taste. See [AbelSuss85] or [Dybvig] for more extensive examples.

8.1 A BRIEF INTRODUCTION TO THE PROGRAMMING LANGUAGE SCHEME

Scheme inherits lexical scoping from Algol and syntax from Lisp. The basic rule for evaluation is to evaluate all sub-expressions between the balanced parentheses and apply the value of the first to the values of the others. There are a few "special forms" which do not evaluate all their arguments. These are noted as they occur.

Examples:

```
                Scheme                  ~C
(<=? lo-bound x hi-bound) ; ((lo-bound <= x) AND (x <= hi-bound))
(+ 23 31 4)               ; (23 + 31 + 4)
((if (foo x) * +) 2 5 3)  ; (foo(x) ? (2 * 5 * 3) : (2 + 5 + 3))
                          ; I.e. the IF construct returns the function to
                          ; be applied to the arguments 2, 5, and 3.
```

Scheme has very few fundamental constructs. E.G.:

abstraction:
```
(LAMBDA (x) (* x x))   ; 'lambda' indicates an unnamed function
(DEFINE (sq x) (* x x))
(DEFINE sq (lambda (x) (* x x)))
```

application:
```
((lambda (x) (* x x)) 2)    ; returns 4
(sq 2)                      ; returns 4
```

conditional:
```
(IF test consequent alternative)    ; IF evaluates consequent XOR alternative
```

assignment:
```
(SET! x 3)
```

sequencing:
```
(BEGIN (set! x 3) (sq x))           ; BEGIN returns the value of the last expression
```

```
; A COMMENT begins with a semicolon and runs to the end of the line.
```

Scheme is extended via syntax transformations. For example LET, which declares local names can be implemented via lambda abstraction:

```
(let ((a 1) (b 2)) (+ a b)) <==> ((lambda (a b) (+ a b)) 1 2)
```

Scheme has excellent abstraction features and supports first class procedures and continuations.

{Continuations represent the default future of a computation and their use allows the creation of advanced forms of control structures used for example to implement coroutines, backtracking and multiprocessing. They have importance in program transformation strategies as well [Clinger87, FriHaKohl84, FriHa85, HaFri87, HaFriWa84, Wand80a, Wand80b].}

8.2 EXAMPLE: Variations on the Greatest Common Denominator (GCD) algorithm

Here are 4 variations on the GCD algorithm. The main point of this series of examples is to show how a functional algorithm (without assignment) can be transformed into an imperative one (without function calls). The secondary objective is to show a number of styles [functional, imperative, continuation-passing, object-oriented] in the same programming language and demonstrate a (trivial) use of first class functions.

8.2.1 Functional GCD algorithm -- no assignment

The first example is functional. The only real function call inside ***GCD-1*** is to ***REMAINDER***. The other calls are in the 'tail' position. What this means in practice is that there is no more work to be done in the function after the last call, so no extra return address has to be passed. The "tail call" can use the same return address that was passed in to the main function and so tail recursive loops can be compiled into simple jumps. Recursive tail calls are just loops.

```
;-------------------------------------------------------

(define (GCD-1 a b) (if (= b 0) a (gcd-1 b (remainder a b)) ) )

(define (REMAINDER n d)
        (if (< n d)
            n
            (remainder (- n d) d)    ; This is a 'tail-recursive' call. This
) )                                  ; code compiles into a simple loop with
                                     ; no recursion.
```

8.2.2 Continuation Passing Style (CPS)

In this second example 8.2.1 has been transformed into Continuation Passing Style (CPS). What is happening here is that, instead of returning a result, each function gets an auxiliary argument, the continuation, which is a function which gets called with the result. The top-level continuation (identity in this case) is the only one that ever returns a result. Essentially, the continuation holds the values which would be on the stack, but in a form which can be manipulated directly. Note that ***CPS-REMAINDER*** just passes along the same continuation in its loop. It "never has to push a new return address."

The advantage of CPS is that it makes all temporary variables explicit. It is done as part of the compilation process by some compilers [AppelJim, Kranz].

```
;-------------------------------------------------------

(define (IDENTITY value) value)  ; just take a value and return it.

(define (GCD-2 a b) (cps-gpd a b identity))
```

```
(define (CPS-GCD a b k) ; k=continuation
        (if (= b 0)
          (k a) ; result is passed to the continuation
          (cps-remainder a b (lambda (v) (cps-gcd b v k)))
) )

(define (CPS-REMAINDER n d k)
        (if (< n d)
          (k n)
          (cps-remainder (- n d) d k)
) )
```

8.2.3 Register Machine with Stack

The third example is worth some study. It was derived from 8.2.2 but is basically code for a 2 register machine with a stack. The function calls are now jumps. A function call is seen as a goto which passes arguments, one of which is the continuation or return address. Note that the 'extra' parenthesis around ***((pop STACK))*** means that the result of popping the stack is a function which is then called. The only real trick between 8.2.2 and 8.2.3 is that the arguments are passed implicitly in registers. [You may have to 'hand execute' the code for a few values to convince yourself that it really works].

The stack abstraction for 8.2.3 is just an example of a simple object in the object oriented sense. Executing ***(make-stack)*** is roughly equivalent to sending a stack 'class' the 'new' message. What is returned is an object (a functional closure in this case) with its own local state (the ***a-stack*** variable). Executing ***(push STACK 5)*** is implemented by sending the 'push' message to the STACK object. What is returned is a function which is applied to the argument, 5, which is added to the stack. There are several object systems available for Scheme (see e.g. [AdamsRees] for implementation details).

```
;-----------------------------------------------

; REGISTERS
(define R0 0) ; result and temporary register
(define R1 0) ; temporary register
(define STACK (make-stack)) ; stack abstraction

; ALGORITHM
(define (GCD-3 a b)
          ; setup initial state & run
          (set! R0 a) ; set up registers
          (set! R1 b)
          ; set R0 as result of computation
          (push STACK  (lambda () R0)) ; just return its value
          (register-gcd) ; jump to initial pc
)

(define (REGISTER-GCD)
        (if (= R1 0)
          ((pop STACK)) ; set PC to addr on stack
          (begin
            (push STACK R1)
            (push STACK gcd-continuation)
            (register-remainder) ; jump to label "REGISTER-REMAINDER"
           )
)       )
```

```
(define (REGISTER-REMAINDER)
        (if (< R0 R1)
            ((pop STACK)) ; execute the saved continuation
            (begin
                    (set! R0 (- R0 R1))
                     (register-remainder) ; jump to label "REGISTER-REMAINDER"
            )
)       )

(define (GCD-CONTINUATION) ; value in R0
          (set! R1 R0)
          (set! R0 (pop STACK))
          (register-gcd) ; jump to label "register-gcd"
)

; STACK ABSTRACTION

(define (MAKE-STACK)              ; --returns a new stack object each time it is called
                                  ; Code is shared by all returned "instances".
        (let ( (a-stack '()) ) ; This list implementation can be replaced by an
                               ; array for speed, with no change in interface.
          (define (stack-push value)
            (set! a-stack (cons value a-stack))
          )

          (define (stack-pop)
            (if (null? a-stack)
                (error "Attempt to pop from empty stack!" '())
                (let  ( (result  (cara-stack)) )
                    (set! a-stack (cdr a-stack))
                    result)
          ) )

          (define (dispatch message) ; message passing style of dispatch
            (case message
              ((push) stack-push)  ; return function matching message tag
              ((pop)  stack-pop)
              (else (error "STACK: Unrecognized message:" message))
          ) )

          dispatch)          ; Return the dispatch function as a new 'stack object' instance.
)

; convert message passing interface to function call interface
(define (PUSH stack value)
          ((stack 'push) value))

;; (stack 'push)  returns a function which is applied to "value"

(define (POP stack)
          ((stack 'pop))) ; execute the stack's internal pop routine
```

8.2.4 The above (8.2.3) as pseudo machine-code:

The fourth and final variant is what a compiler might emit as a result of compiling the functional definition 8.2.1 and inlining remainder. It is a straight forward transliteration of 8.2.3 into pseudo-code (and the only "code" shown which has not been executed on a computer).

```
;-------------------------------------------------
*
* START
*
GCD:    * ENTRY SETUP -- assume R0 has A and R1 has B at call
        store   @END-GCD,++(SP)     * push code address to come back to
        jump    REGISTER-GCD        * go do the work
END-GCD:
        rts     * return-from-subroutine (comes back indirect)
*               * upon return, R0 has result
*
REGISTER-GCD:
        zero?   R1
        brFalse L1
        load    PC,(SP)--           * pop from stack to PC -- indirect branch
L1:
        store   R1,++(SP)           * push R1
        store   @GCD-CONTINUATION,++(SP)* push label address (where to continue)
*       jump    REGISTER-REMAINDER      * fallthrough
*
REGISTER-REMAINDER:
        less?   R0,R1
        brFalse L2
        load    PC,(SP)--           * pop from stack to PC -- indirect branch
L2:
        sub     R0,R1               * result in R0
        jump    REGISTER-REMAINDER
*
GCD-CONTINUATION:
        regmov  R1,R0               * R1 <- R0
        load    R0,(SP)--           * stack value pop'ed to R0
        jump    REGISTER-GCD
*
* END
*
```

SUPPORT FOR QUALITY IN THE ASPIS[1] ENVIRONMENT

Dag Mellgren
CAP SESA Innovation
Centre de Recherche de Grenoble
33, Chemin du Vieux Chêne, ZIRST
38240 Meylan, France
Tel. (+33) 76908040
E-mail : uunet!mcvax!csinn!mellgren

ABSTRACT

The ASPIS project has built active tools (*assistants*) that support the early stages of software development. These assistants are based on artificial intelligence techniques and exploit knowledge about the development methodology and the application domain. They are able to perform semantic as well as syntactic checks on analysis and design documents. This is made easier by the internal representation of the documents in an object-oriented knowledge representation system. Software metrics on the design language have also been defined, providing a quality measure. An effective communication between the two main development assistants, the Analysis Assistant and the Design Assistant, contributes to bridging the gap between requirements and design. Prototyping capabilities and other features provide short-cuts with respect to the traditional software life-cycle.

KEYWORDS

software development, analysis, design, reusability, prototyping, CASE, assistant, domain knowledge, method knowledge, knowledge-based system.

Author's biographical sketch

Software engineer at CAP SESA Innovation (research branch of the CAP Gemini Sogeti group), since June, 1988. Prior to this, worked for Teknowledge, Inc. (Palo Alto, CA), in the expert systems business. Received an engineering diploma from the Royal Inst. of Technology in Stockholm, Sweden (1984) and a M.S. degree in Computer Science from Stanford University (1985).

---

1. ESPRIT project 401, Application Software Prototype Implementation System. Partial funding provided by the Commission of European Communities as part of the European Strategic Programme for Research in Information Technology (ESPRIT).

1. INTRODUCTION

Today tools have been built to support most of the popular analysis and design methods such as JSD [Jackson, 83] or SADT [Ross, 77]. Software engineering shells, in which the user can describe his favorite methodology, also exist on the market. On the other hand, starting with the MIT's Programmer's Apprentice [Waters, 82], both research and commercial projects have used artificial intelligence techniques to build assistants for software developers. However, these often focus on the coding task, and formal design methods are not necessarily supported.

The ASPIS environment combines analysis and design methods with AI-based intelligent support. It contains four assistants that will help with the specification and design phases of software development. We do not address the coding task directly, rather we concentrate on the early part of the life-cycle, critical to the quality of the system. We contribute to making maintenance easier by helping to produce a careful analysis and design.

The key feature of the ASPIS assistants is their knowledge about the methodology and the application domain. Method knowledge enables the assistants to follow a given method, and to provide help about it to the user. Domain knowledge allows the assistants to supply expert advice, for types of application that are sufficiently well-understood. Thus, for instance, a designer who is not familiar with an application domain can benefit from domain-specific suggestions. Because method and domain information is stored in replaceable knowledge bases, the assistants are independent of any particular method or domain. As a comparison, another ESPRIT project, DAIDA [Jarke, 88], has created a knowledge-based environment for developing database-intensive systems. It also exploits domain knowledge, but both the methodology and the knowledge representation are tailored to one type of application. A design tool for accounting applications, developed in the United States, is described in [McCarthy, 89]. Other projects such as KDA (Knowledge-based Design Assistant) [Sharp, 88] use method rather than domain knowledge. In our view, for well-known applications, knowledge about the domain is more powerful than knowledge about the discipline of programming and designing in general (see also [Adelson, 85]).

A general overview of the ASPIS project may be found in [Hughes, 88]. This paper focusses on the features of the ASPIS assistants relevant to quality assurance. After a brief description of the four assistants and their interaction, some general environment features are shown. Next, the communication between the Analysis and Design Assistants is presented in more detail. Then, we explain the available checking features and software metrics. Finally, we conclude by discussing next-generation CASE tools.

2. THE ASPIS ENVIRONMENT

The ASPIS environment contains a set of cooperating assistants: the Analysis Assistant (AA), Design Assistant (DA), Prototyping Assistant (PA) and Reuse Assistant (RA). Prototypes of these assistants were last demonstrated at ESPRIT Technical Week (Brussels, Nov. 88). They run on Sun-3 workstations and are implemented in Quintus Prolog [Bowen, 85]. We also use an object-oriented Knowledge Representation System developed by the project (for a short overview, see [Hughes, 88]) and Quintus ProWindows [Quintus, 88] for the graphical user interface.

The figure below shows the ASPIS assistants and their interaction.

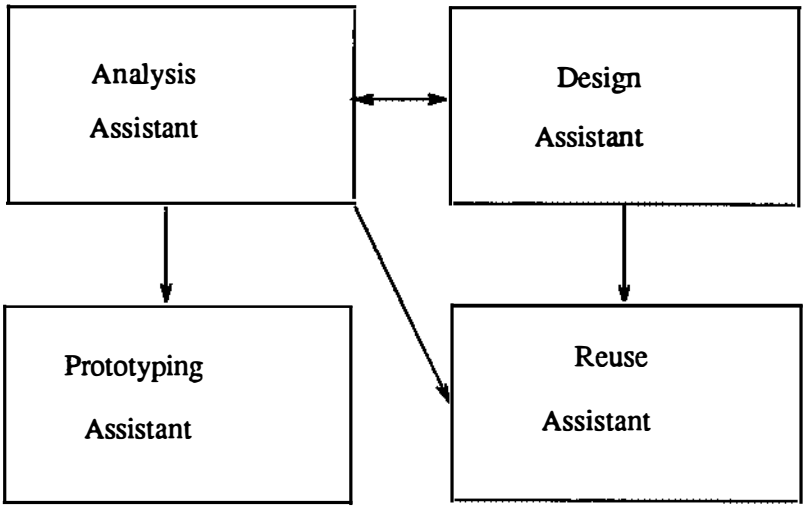


Fig. 1 - The four assistants

Since ASPIS is based on domain knowledge, we had to choose application domains in which to work, although it has to be stressed that the underlying techniques are general. Because of resource constraints on the project, the ASPIS prototype currently works in a single domain: physical access-control systems. These systems regulate access to a site via the use of monitoring devices such as card readers and alarm sensors. This domain was chosen because

- Access control systems are embedded systems that also include some data processing aspects.
- The problem is easy to visualize for demonstration purposes
- Expertise was readily available from one of the partners in the project

2.1 **The Analysis Assistant (AA)**

The AA supports requirements analysis and functional specification. The method used is based on Structured Analysis (SA) [Ross, 77] and the entity-relation (ER) model [Chen, 76]. The analysis documents produced with the AA are either:

- SA diagrams describing the functions of the system
- ER schemata representing the data
- Non-functional requirements, such as size of the system, cost, availability, or security. These are represented as structured texts.

These constraints are then taken into consideration by the Design Assistant.

2.2 The Design Assistant (DA)

The Design Assistant covers the hardware as well as the software aspects of design. It also distinguishes between general and detailed design phases. The software design method used is a top-down approach matching the structure of AMPHI, a design language developed within the ASPIS project (Abstract Machine and Process Hierarchies). AMPHI is inspired by a paradigm and a language based on abstract machines: MACH [Galinier, 85], but extended to offer a hierarchy of processes. This is useful when designing real-time systems. The syntax of the AMPHI design language is reminiscent of Ada. The hardware design is described using a hardware description language allowing a hierarchy of subsystems and individual components of different types such as processor, disk or cable.

2.3 The Reuse Assistant (RA)

Being able to re-use designs or code is an important means of saving time and increasing the granularity of the subtasks to be performed. The base of reusable components is a natural extension of the domain knowledge base. The ASPIS Reuse Assistant works together with the Analysis and Design Assistants in order to be able to reuse both analysis and design documents. Through pattern-matching techniques on the functionalities and interfaces of the SA boxes (analysis) or abstract machines (design), the RA is able to propose a set of potentially useful components. A key feature of the RA is that it keeps an eye on successive refinements in the analysis or design. As more information becomes available, it is able to reduce its working set of matching components incrementally.

By using the RA together with one of the development assistants (AA or DA), the user of ASPIS can put together existing elements and produce a detailed analysis and design of an application. This could be combined with automatic programming techniques: if you can reuse detailed designs, you might be expected to be able to reuse the corresponding code. A programming tool could be added to ASPIS to collect pieces of code and assemble them in order to produce a final program.

2.4 The Prototyping Assistant (PA)

A formal specification language, called Reasoning Support Logic, has been developed within the ASPIS project. The SA diagrams may be annotated with properties expressed in this language. The Prototyping Assistant allows the user to verify the consistency of the specifications by supporting animation of the SA diagrams with annotations.

3. ANALYSIS AND DESIGN METHODS

The different steps in the methodology enforced by the Analysis and Design assistants are considered as part of their method knowledge. The assistants keep track of which steps are left, ongoing or finished. They know about the criteria for each step to be finished. Below we give an overview of the major steps in analysis and design, respectively.

3.1 Analysis

The Analysis Assistant supports both functional and data analyses using the SA and ER formalisms, respectively. After both analyses have been completed, the AA provides a facility to help the analyst compare the system data descriptions given in SA and ER. Through the exploitation of heuristics to derive information from the structure of the data included in the SA and ER documents, possible

inconsistencies can be found by the AA between the two descriptions.

A natural extension of this method, not implemented in the current prototype of the Analysis Assistant, can be derived from [Feather, 89]. It consists in describing the system in parallel from different points of view. This allows the complexity of the system specification to be broken down and the analysis to be carried out from different perspectives, thus leading to a better insight. In addition, as several descriptions are produced for common aspects, consistency and completeness checks can be performed on the intermediate results. The final step of this procedure is to integrate the different viewpoints, yielding a final set of analysis documents.

The Analysis Assistant also lets the user enter a number of non-functional requirements. For instance, the user can specify that the system should have very high availability, or that software security should be high. These requirements are later used by the Design Assistant. They can refer to both the software and the hardware architecture of the system (typically the hardware).

3.2 **Design**

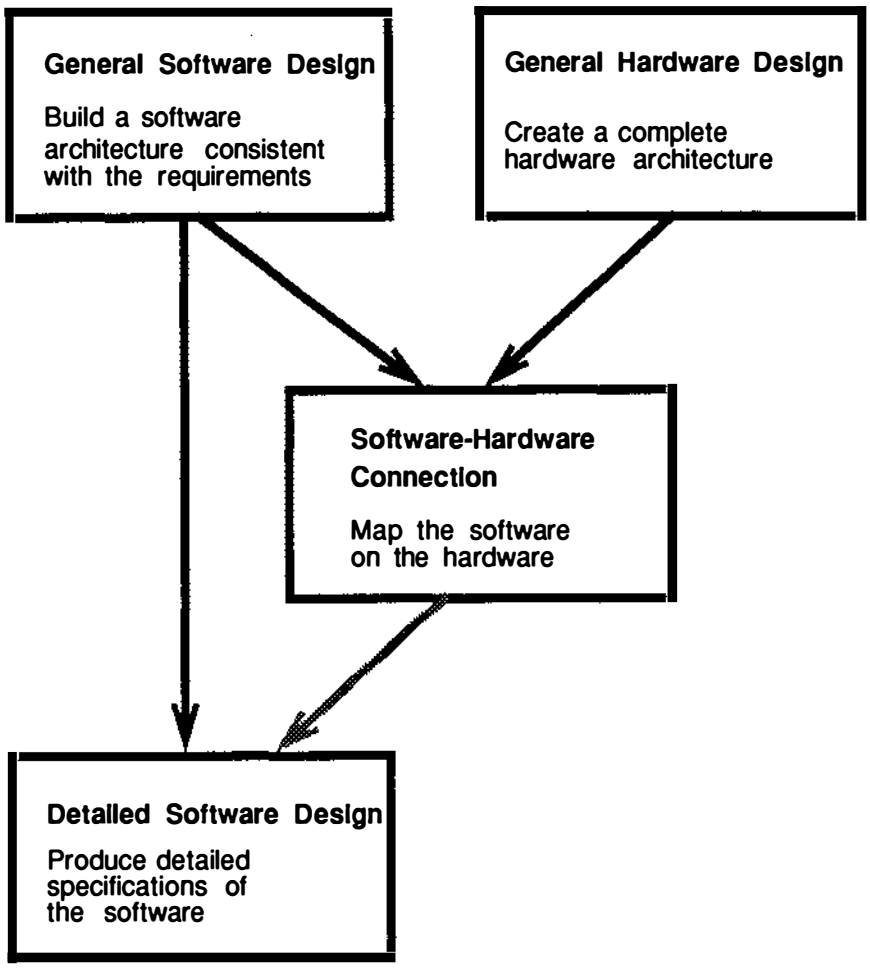


Fig. 2 - The design process

The design activity is decomposed in four main phases: general software design, general hardware design, software-hardware mapping, and detailed software design. These design phases and the precedence between them is shown in the diagram above.

The distinction between general and detailed software design is the following:

- In general software design, only the interfaces of software components are specified (imported and exported resources, events and operations).
- In detailed software design, we reach the point of specifying pseudo-code for processes and operations.

The objective of the hardware design phase is to get a hardware architecture described in enough detail to see how it supports the software. We are not interested in hardware design *per se*. We need to know about the functionalities provided by the hardware architecture, but not the details of how they are implemented. Therefore, there is no need for a detailed hardware design phase.

The software-hardware mapping is an important phase as it occurs after the software and hardware design phases and it validates the entire system. Its purpose is to establish links between hardware and software components, e.g. which processor a process runs on, or on which device an abstract machine is stored.

4. GENERAL FEATURES

4.1 Translation

The purpose of the *translation* facilities of the Design Assistant is to create initial designs from the requirements specified with the Analysis Assistant. This has several advantages:

- The designer gets started rapidly and can concentrate on refining a rough design instead of starting from scratch.
- As the output of the Analysis Assistant is directly used as input to the Design Assistant, there is a guarantee that all requirements are taken into account.

There are two separate translation facilities in the Design Assistant: one for software design and one for hardware. The translations are performed interactively with the user, using domain knowledge to make suggestions.

The software translation takes as input the SA and ER diagrams from the analysis phase and constructs a set of programs, processes, machines and operations described in the AMPHI language, and conforming to the requirements.

Fig. 3 is a screen copy showing an example of suggestions that are made in this phase. It is taken during the translation of the SA diagram "A-0" for an access-control system. A-0 is the root of the refinement tree of SA diagrams, and it represents the whole system as a black box with inputs and outputs.

For the A-0 diagram, all input and output arrows are automatically translated into events, and the user only has to specify their origin or destination (an I/O module). The assistant displays, for each

Fig. 3 - Hard copy: DA during software translation

arrow, the suggested origins or destinations. There may be several suggestions. The first one is pre-selected as the default value in the window on the upper right-hand side. For instance, the output 'reports' will normally go to either the program 'printer' or the program 'terminal', but 'terminal' is favored.

To make these suggestions, the assistant examines its domain knowledge and finds a list of possibilities for each one of the arrows. (If there is no knowledge, the list is empty.) This list is then ordered according to the following criteria (in increasing order of priority):

1. If the suggestion is a "mechanism arrow" in the SA diagram, then bump its priority. (Mechanism arrows are arrows that specify support needed by the box to perform its function.)

2. A module already present in the design has priority.

3. If it is the current module in the DA's memory, then it has the highest priority.

The reason why there can be already existing modules is that these may either have been created manually, or they may have been introduced in a previous translation. Several translations may be needed, in particular when requirements change.

Once the list is sorted, it is displayed by order of priority. For instance, the event 'access request' can come from 'card reader' or 'key pad', where the first is more likely.

The A-0 diagram is actually a special case of the software translation. In general, the user will have to determine which arrows in the SA diagrams should become real-time events, and which should merely be translated into resources (data). (Again, some suggestions can be made in this step from domain knowledge.) Then, the DA will translate into processes those SA boxes that send or receive at least one event. The other boxes become operations to be grouped within machines. The DA also has knowledge about how to group the operations into known machines. As usual, the designer is completely free to reject any advice and we do not think it is a good idea to force him into set solutions.

We now describe the hardware translation. It takes as input the mechanism arrows of the SA diagrams (which often represent hardware requirements), and the non-functional requirements. Among these, the size requirement specifies how many devices of different kinds are present. Based on this, and the physical distribution of the system, the Design Assistant suggests some basic hardware architectures (e.g. single processor or distributed architecture). A small expert system helps the designer determine the best basic architecture for his needs, and also suggests some enhancements that affect positively the other non-functional requirements.

4.2 **Reusability**

As we have seen in the previous section, suggestions are made automatically during the *translations*. This is generally the case in the ASPIS environment, where the tools are active to some extent.

Possibly the most important suggestions are those made by the Reuse Assistant, which communicates with both the Analysis Assistant and the Design Assistant. Reusability is thus provided at two levels: for analysis documents and design documents.

When using the AA, the user is presented with a set of reusable components. These are found by searching through a base of components. The matching algorithm is based on the semantic of SA

boxes and their inputs and outputs. The semantic is defined as a canonical name for the box, and the generality of this approach is ensured by a synonym mechanism. Domain terminology and synonyms form an important part of domain knowledge.

In a similar fashion, when editing design documents, the user will be given the opportunity to reuse existing designs if they match the specifications, and there are not too many candidates. Matching operates on the semantics of the software components and their interface.

When the designer is using the DA's editor of software components, a REUSE button will appear in the window when reusable components are available. More precisely, when there is at least one but less than a certain number (say 5) of potentially reusable components. When the user clicks on the REUSE button, the RA retrieves the components. There is a browser to examine them, and the possibility of integrating a selected reusable component with what was being edited, thus completing the reuse operation.

4.3 **Prototyping**

The Prototyping Assistant supports the creation and animation of a formal model of the functional specification. (In the ASPIS prototype, the functional specification is given by the SA diagrams, but the Prototyping Assistant could easily work with any other language specifying data flow among activities.) Tools are provided to

- create the formal model in our logic-based formal specifications language (Reasoning Support Logic). This language allows the developer to state properties of the data involved in a system and rules for specifying data flow dealing with both structural and - especially - timing constraints.

- checking and compiling the model into an executable form

- interpreting the executable model (animation). The result of animation can be printed textually or shown graphically by highlighting the data paths on the SA diagrams.

In RSL, a so-called 'position triple' specifies the value of a certain datum at a certain time. The form is "Data val Value at Time", where Data, Value and Time are variables. An example from our access-control domain would be "access request val Request at Time_card_inserted". Position triples can be grouped to form 'states'. Each state, in addition to a set of position triples, contains a time stamp, which represents the time at which the state is current.

The animation takes place in two stages. Initially a query must be obtained from the user to initiate animation of the specification. The query is then compiled to check its syntax and to convert it into its internal representation. The compiled query is then passed to the RSL executor for animation. The query specifies an initial 'state' for the values of the data in the system. From this initial state, we try to work out later states by using the executable model and attempting to find a solution to the query. If a solution can be found then it is presented to the user by highlighting the SA boxes and data that were involved in the solution on the SA diagram. If any external data was produced in the solution then the values and times of these data items are presented to the user in a window. From this graphical presentation the data flowing through the system and the functions used in the system can be traced.

Thus, the Prototyping Assistant allows both the user and the analyst to validate the application specifications before design. This provides a useful short-cut in the traditional life-cycle. An improvement on the current ASPIS prototype would be to offer rapid prototyping capabilities at the

design level as well.

5. AA-DA COMMUNICATION

5.1 Marks

The communication between the Analysis and Design assistants is one of the main features of ASPIS. It mimics the communication between the human analyst(s) and designer(s). Its purpose is to bridge the gap between analysis and design, making sure that requirements are fulfilled.

Because analysis usually precedes design, the DA is meant to be invoked after the AA, and it exploits the analysis documents already produced. In order to do this, the DA reads a file written by the AA containing SA, ER diagrams and non-functional requirements. (It can still run if no requirements are present.)

In addition, the AA and DA communicate both ways through the *marks* mechanism. Conceptually, these marks are annotations, made with one assistant, on the documents produced with the other assistant. These marks can be set either manually or, in some cases, automatically by the assistant. There are several different types of mark:

- **used:** means that one or more analysis documents were used by the DA to produce certain design documents. The mark contains pointers to both the analysis and design documents concerned.

- **Incomplete:** set by the DA on a given analysis document where some critical information was missing.

- **Incompatible:** some requirements were found to be contradictory during the design phase. For instance, the DA may set an incompatible mark pointing to two different non-functional requirements, such as cost and security. The reason is that attempts to satisfy both constraints simultaneously failed.

- **changed** or **deleted:** the AA signals to the DA that some requirements were changed or deleted, respectively, since the last time the DA was invoked. This prompts the DA to review design documents that were produced using these requirements.

These marks are acknowledged by the assistant concerned (by setting a taken_account flag) and passed back to the other assistant.

As an example, let us imagine that non-functional requirements on the cost and the security of the system have been defined by the analyst. When the hardware translation is run, these are taken into account and this is signalled to the AA by a **used** mark. Possibly, the cost requirement was too tight for the security requirement to be fulfilled. In this case, the DA sets an **Incompatible** mark and sends it to the AA. The AA informs the analyst that one of these requirements needs to be modified because of the **Incompatible** mark. The analyst relaxes the security requirement. Because a **used** mark was set on this requirement, the AA sets a **changed** mark to inform the DA that a requirement was changed. Then the designer has to perform some re-translation. The DA sets the **changed** mark taken into account, and the new requirement is marked **used**. This sort of procedure could go on for several rounds before everything is resolved.

5.2 Traceability

A perfect *quality tag*, for any product, would include its entire history. The ASPIS Design Assistant automatically maintains pointers to analysis documents within design documents. This is done by the so-called *software translation* facility (section 4.1) which interactively transforms specifications into an initial design. The *marks* produced during the development are useful to keep track of the history of modifications in analysis and design.

The *history* and *viewpoint* facilities available in the Design Assistant are also relevant. History keeps track of steps and operations performed in a fairly great level of detail (creation or modification of a software or hardware component). The viewpoint mechanism lets the designer keep a tree of intermediate states of the design, where every branch corresponds to some design decisions. The designer may then change context and go back to any such state (*viewpoint*). In particular, going back to the parent viewpoint means undoing the modifications that were performed since that viewpoint was saved.

6. CHECKING AND EVALUATING ANALYSIS OR DESIGN DOCUMENTS

6.1 Checking Features

Checking facilities, capable of detecting certain classes of errors immediately, are provided by both the Analysis and Design assistants. The mistakes covered by these facilities might be costly if they were allowed to remain in the design documents and not be detected until later in the life-cycle.

In the Analysis Assistant, the domain knowledge affords a number of checks on the SA decomposition of the system. For instance, Figure 4 shows the state of the SA diagram editor for the A-0 (root) diagram of an access-control system. The inputs to the system are 'security controller input' and 'access request', the outputs 'reports' and 'alarm', and the control 'data storage formats'.

The warnings are displayed in the window in the lower right corner. The assistant has noticed that there is no input to the box whose semantic (i.e., canonical name according to the synonym knowledge) is 'alarm stimulus', input from the sensors. That could be an important omission. Similarly, there is no control arrow with the semantic 'time'. Therefore, the system's behavior will not vary with the time of the day. This should probably be the case. Both these examples correspond to one class of mistake: "one arrow missing from a given box". The three next warning messages correspond to the class of mistake "one arrow present, but a related arrow missing". To summarize, these error messages illustrate two types of common mistakes in the AA's domain knowledge.

In the Design Assistant, checks are provided both for the software and hardware architecture. They are divided into consistency and completeness checks. An example of completeness check is verifying that all the interfaces of the software components have been specified (this also signals the end of the general software design phase). Furthermore, one can separate outright errors from warnings on the style of the design. An example of the latter class of flaws is having too many hardware components within a single subsystem. The maximum number of components can be defined for each project, and whenever it is exceeded, a warning will be displayed.

The boundary between passive "checks" and active "suggestions" is not clear-cut in ASPIS. When the command to perform checks is issued, some suggestions on how to improve the design are also in order. For example, in the software-hardware mapping phase, if we know that we have high security requirements and a distributed architecture, we are able to suggest that all communications should be encrypted.

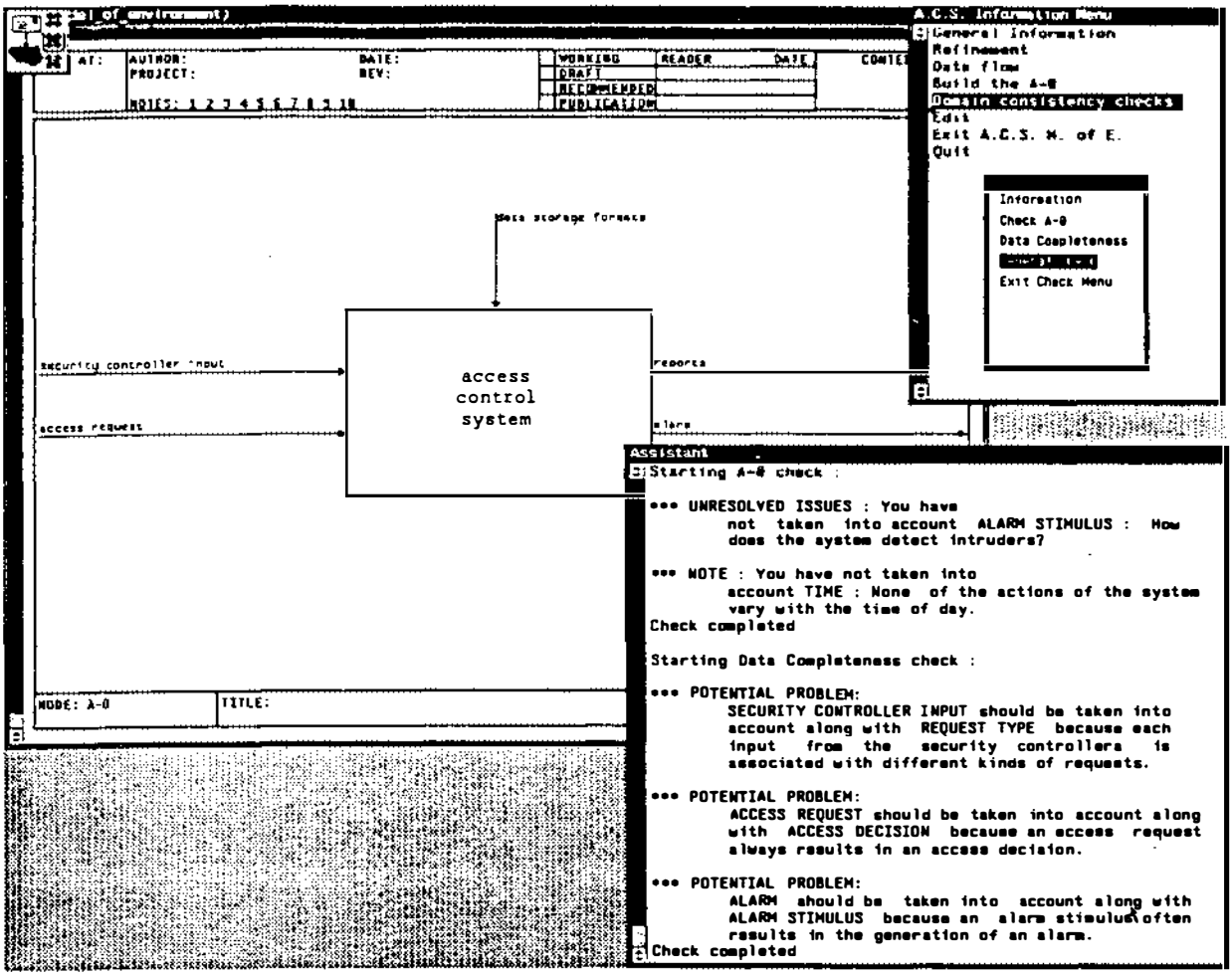


Fig. 4 - Checks in the Analysis Assistant

Another, more direct type of check is performed by the Software and Hardware editors. These editors are coupled with parsers for the software and hardware design languages (AMPHI and HDL, respectively). Thus, the syntax is checked whenever the designer saves modifications to the design documents. The description in AMPHI or HDL of the component is parsed into the internal representation of the ASPIS design documents, which is in an object-oriented knowledge representation system developed by the project. This representation facilitates checking of semantical properties rather than purely syntactical. For instance, if the designer is editing a process and adds a sub-process that is already a sub-process of another process, this will be detected and a warning message displayed. This is because each process must have a unique parent. Similarly, the editor will detect if an operation is being placed into more than one machine, and warn the user when this is the

case.

6.2 Software Metrics

In addition to detecting outright errors, software metrics can be useful to detect flaws or check some rules about the design. The Design Assistant implements the following types of measures on AMPHI descriptions:

Modularity

> These measures are based on the dependency graph of abstract machines. A machine is said to depend on another when it imports operations or resources (data) from it. As a rule of thumb, resources should be imported only from machines directly below the given machine in the dependency graph. There are exceptions, such as machines offering a library of primitives. In this context, we can also detect unnecessary exports, and suggest that the designer split machines that are used by a lot of other machines.

Complexity

> These metrics work on the pseudo-code specified for the algorithms corresponding to processes or operations within machines. Several measures are performed, such as:
>
> — McCabe complexity [McCabe, 76]
>
> — Number of instructions
>
> — Number of comments
>
> — Number of literals
>
> — Maximum block nesting level
>
> — Number of goto statements
>
> Rules to warn the user based on these measures can be customized for each project.

Machine cohesion

> An abstract machine groups operations that depend on the data structure of the machine. Thus, each machine represents an intermediate level of abstraction in the design process. As a rule of thumb, each operation in a machine must, directly or indirectly, consult or modify the data structure of the machine. If this is not the case, the machine should be split or merged with other machines.

7. CONCLUSION

We believe that ASPIS and other projects have demonstrated that knowledge-based techniques provide an effective means of tackling the task of systems development. In particular, there are many advantages of the application of domain expertise. It enables less experienced users to learn how the domain experts work. At the same time, it unloads these experts so that they can make better use of

their time. Thus, we think that the next generation of CASE tools will incorporate knowledge-based features. This should contribute to resolving the software crisis.

ASPIS also contributes towards the goal of achieving reusability of software, which would be a major step in increasing quality. Reusability remains a research topic, but will become useful at least in some application domains.

ACKNOWLEDGEMENTS

This paper presents work by the entire ASPIS team, at Tecsiel SpA (Pisa, Italy), GEC Marconi Software Systems (Borehamwood, England), GEC Marconi Research Centre (Chelmsford, England), LGI at the University of Grenoble (Saint Martin d'Heres, France) and CAP SESA Innovation (Meylan, France).

TRADEMARKS

SADT is a trademark of SofTech (in the USA) and IGL Technologie (France).

JSD is a trademark of M. Jackson.

DAFNE is a trademark of Italsiel and CNR.

Sun-3 is a trademark of Sun Microsystems, Inc.

Quintus Prolog and ProWindows are trademarks of Quintus Computer Systems, Inc.

MACH is a trademark of IGL Technologie

Ada is a registered trademark of the U.S. Department of Defense

REFERENCES

[Adelson, 85] Adelson, B. and Soloway, E.: "The role of Domain Experience in Software Design", IEEE Transactions on Software Engineering, special issue on Artificial Intelligence and Software Engineering, vol SE-11, Nov. 85.

[Bowen, 85] Bowen, D. and Znidarsic, D. "Quintus Prolog Reference Manual", Quintus Computer Systems Inc, Mountain View, CA, 1985.

[Chen, 76] Chen, P.: "The Entity-Relationship Model: Toward a Unified View of Data", ACM Transactions on Data Base Systems, vol. 1, no. 1, Mar. 76.

[Galinier, 85] Galinier, M. and Mathis, A.: "Guide du concepteur MACH", Thomson-CSF, DSE IGL Technology, 1985 (in French).

[Hughes, 88] Hughes, A., Puncello, P. and Pietri, F.: "Intelligent Systems Development in the ASPIS Environment", Proc. 5th Annual Esprit Conference, vol. 1, pp. 380-391. Brussels, Belgium, Nov. 88, North-Holland.

[Jackson, 83] Jackson, M.: "System Development", 1983, Prentice-Hall.

[Jarke, 88] Jarke, M. et al.: "The DAIDA Environment for Knowledge-Based Information Systems Development", Proc. 5th Annual Esprit Conference, vol. 1, pp. 405-422. Brussels, Belgium, Nov. 88, North-Holland.

[McCabe, 76] McCabe, T.J.: "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, no. 4, Dec. 76.

[McCarthy, 89] McCarthy, W. E. and Rockwell, S. R.: "The Integrated Use of First-Order Theories, Reconstructive Expertise, and Implementation Heuristics in an Accounting Information System Design Tool", Proc. 9th International Workshop on Expert Systems and Applications, Avignon, France, May 89.

[Quintus, 88] Quintus Computer Systems, Inc., Mountain View, CA: "Quintus ProWindows User's Guide.", SunView(TM) Version, 1988.

[Ross, 77] Ross, D. T.: "Structured Analysis (SA): A Language for Communicating Ideas", IEEE Transactions on Software Engineering, vol. SE-3, Jan. 77.

[Sharp, 88] Sharp, H.: "KDA -- A Tool for Automatic Design Evaluation and Refinement Using the Blackboard Model of Control", Proc. 10th International Conference on Software Engineering, Singapore, Apr. 88.

[Waters, 82] Waters, Richard: "The programmer's apprentice: knowledge-based program editing", IEEE Transactions on Software Engineering, no. 1, 1982.

AUTOMATING THE SOFTWARE QUALITY PROCESS

ROGER J. DZIEGIEL, JR
ROME AIR DEVELOPMENT CENTER
SOFTWARE ENGINEERING BRANCH (COEE)
GRIFFISS AFB, NY 13441-5700

ABSTRACT

Systems being built today and in the future must be supportable and reliable. Resources, in terms of funds and trained personnel, are fast approaching limits. To alleviate these problems, systems will have to be developed with a focus on quality.

This paper will bring together the research done at RADC to develop techniques and tools to support software quality. The approach taken centers around a framework consisting of 13 software quality factors which are high level user attributes. These factors are then decomposed into a criteria level which are software attributes and then decomposed into metric level questions. A method has been developed to specify, measure and assess software quality using the framework as the foundation. RADC's method to specify and evaluate quality gives insight into problems that are occurring earlier in the development process so that they can be fixed and corrected cost-effectively. The information collected should raise questions into why these problems are occurring in real-time instead of after the project is completed.

BIOGRAPHICAL SKETCH

Roger J Dziegiel, Jr is a computer engineer at Rome Air Development Center (RADC) working in the area of software and system quality. Presently finishing an MS in computer engineering from Syracuse University. He is a member of ACM, and IEEE Computer Society.

AUTOMATING THE SOFTWARE QUALITY PROCESS

ROGER J. DZIEGIEL, JR
ROME AIR DEVELOPMENT CENTER
SOFTWARE ENGINEERING BRANCH (COEE)
GRIFFISS AFB, NY 13441-5700

1.0 Introduction

Software is an increasingly essential and expensive element of DoD defense systems' acquisition and life cycle support. Systems developed not only exceed costs and take longer to build; but the systems, once fielded, are unreliable and/or difficult to maintain. There are increasing quality restrictions placed on software systems today. These restrictions may be vaguely stated as overall goals, and may not be clear to key personnel in the software development process, especially during the early life cycle phases. What is needed is to specify and design quality requirements and measure compliance during the development process instead of testing for vaguely stated quality goals after the software has been integrated to see if it will meet its intended function. This paper will focus on quality and address the tools used to automate the software quality engineering process based on a methodology developed by Rome Air Development Center (RADC).

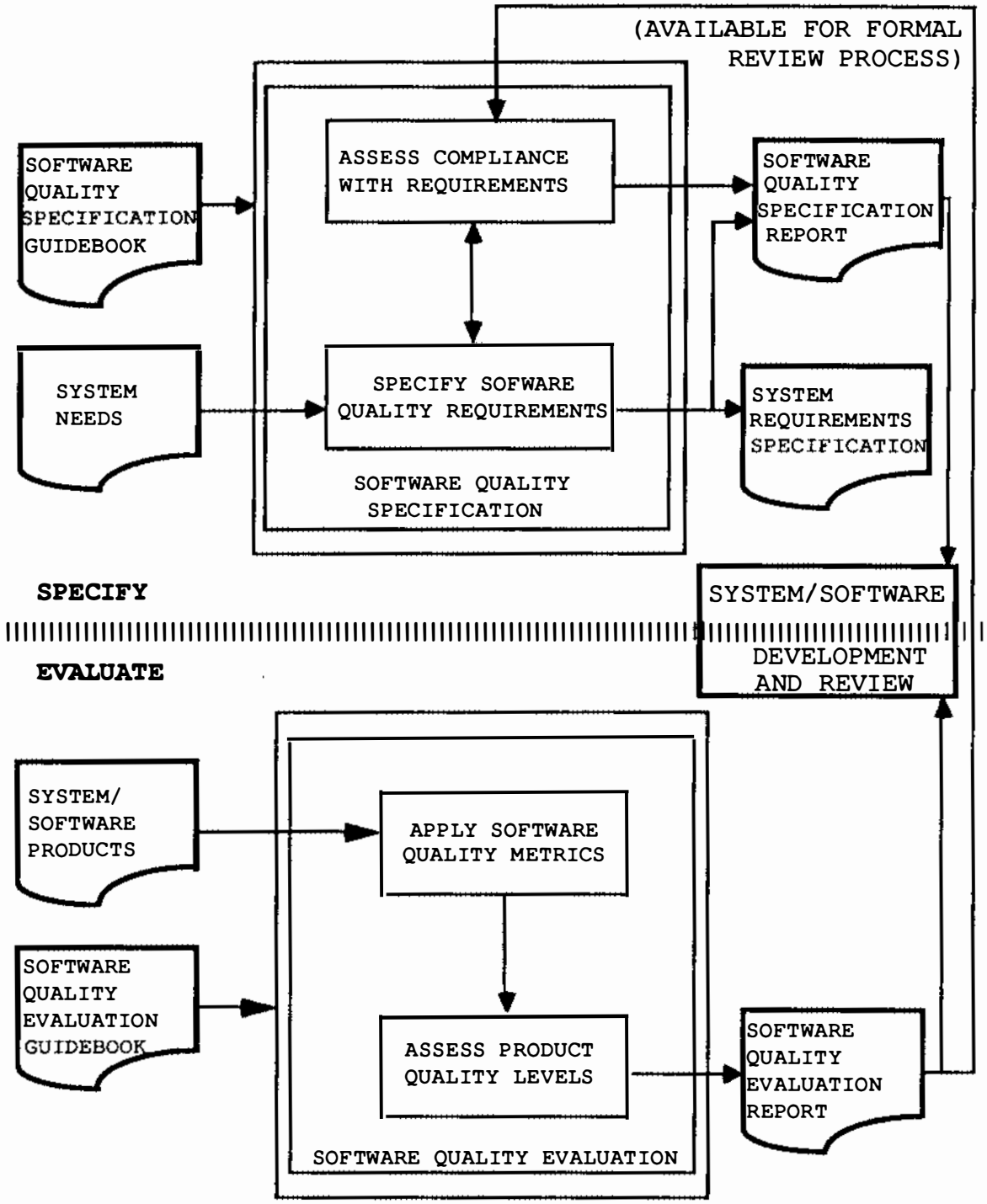
2.0 Background

RADC has recognized that software qualities have to be prioritized/specified, predicted, managed, and evaluated to reduce a software system's life cycle costs. Increased complexity and mission criticality of computer software components in the DoD environment, coupled with higher development costs, have resulted in the need for more emphasis being placed on measuring the quality of software products during all phases of the software development process.

Since the middle 70's, RADC has pursued a comprehensive program aimed at reducing the subjectivity inherent in software quality metrics. This program has resulted in a quality framework and methodology for specifying and evaluating software quality. In addition, development of individual tools to specify, and collect and assess software quality at each phase of the software development life cycle and development of baselines for examining the reliability of software systems are integral parts of RADC's software quality program.

2.1 Methodology

Figure 1 shows the methodology in two major parts: software quality specification and software quality evaluation. Specification is the responsibility of software acquisition managers and includes specifying software quality requirements and assessing compliance with those requirements. Evaluation is the responsibility of the data collection and analysis personnel and includes applying software quality metrics to products of the development cycle, assessing product



THE SOFTWARE QUALITY SPECIFICATION METHODOLOGY ALLOWS ACQUISITION MANAGERS TO CONTROL SOFTWARE QUALITY

FIGURE 1

quality levels, and reporting results. In other words, the quality goals provide the baseline against which to evaluate quality, and the target of convergence for the iterative process of evaluating and improving quality.

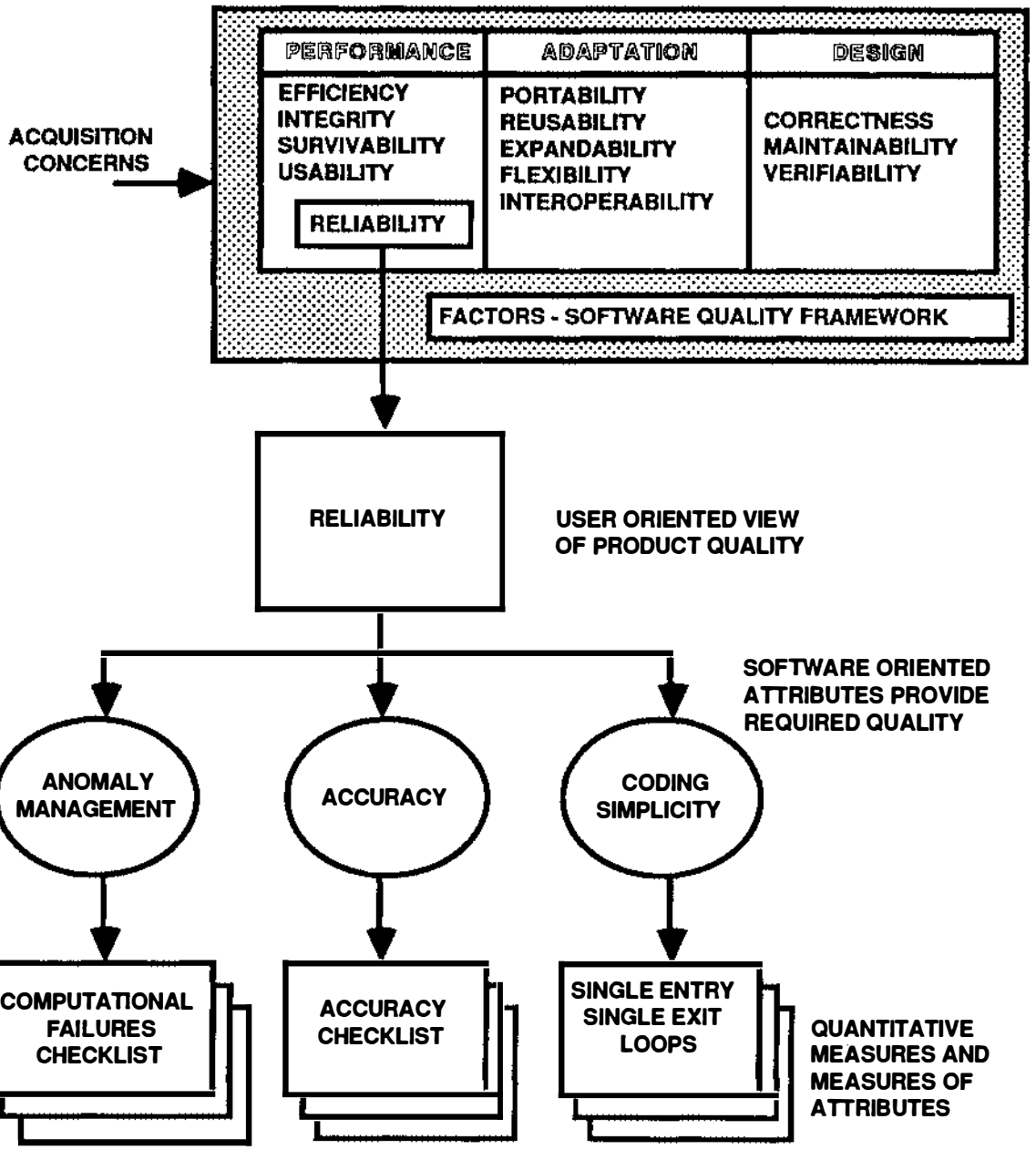
The RADC sponsored development of a software quality methodology to aid in specifying and evaluating software quality which resulted in two guidebooks. The Software Quality Specification Guidebook Volume II, describes the method for specifying software quality requirements and addresses the needs of the acquisition manager. The Software Quality Evaluation Guidebook Volume III, describes the method for evaluating achieved quality levels of software products and addresses the needs of data collection and analysis personnel (Volume 1 is an overview of the total process). The quality model shown in FIGURE 2, in which a hierarchical relationship exists between a user-oriented quality factor at the top level and software-oriented attributes at the second and third level, serves as the foundation for quantifying the presence, absence or degree of identifiable software characteristics. Individual metric elements are combined into metric scores, metric scores into criteria, and criteria into quality factors.[1]

2.2 Tools

In addressing the Software Quality Specification Guidebook, a prototype called the Assistant for Specifying the Quality of Software (ASQS) was completed in 1987. ASQS serves the purpose of transitioning the Guidebook and associated quality engineering method into use in the DoD acquisition process. ASQS bridges the gap between software quality concepts and associated terminology, and system needs and associated terminology understood by DoD acquisition managers. ASQS is an expert system that contains the information necessary to specify software quality requirements.[2]

In 1984, RADC awarded a contract to develop an Automated Measurement System (AMS). The AMS is the automation of Volume III of the Specification of Software Quality Attributes - Software Quality Evaluation Guidebook. The AMS has been designed to implement the framework during all phases of full scale development and automate the data collection and storage process by collecting data from requirement statement languages (RSLs), software design languages (SDLs), and Fortran and Ada source code. Once the data has been collected, analysis can be performed and reports generated. Also, the AMS has the capability to store Software Change/Error data and Resource Expenditures data in the same project database.[3]

At present both the ASQS and AMS are being enhanced. ASQS will include all 13 factors, contain DoD-STD-2167A terminology and contain functional decomposition for the various mission areas (i.e., Armament, Avionics, Command, Control, and Communications (C^3), Missile/Space, and Mission/Force Management) in the knowledge-base. AMS is also being enhanced and will be called the QUality Evaluation System (QUES). It will be enhanced to include more features of Ada, be integrated into the Software Life Cycle Support Environment (SLCSE) or as a standalone system, have more graphical reports and contain DoD-STD-2167A



QUALITY FRAMEWORK REPRESENTATION

FIGURE 2

terminology. SLCSE is a software development environment with a common user interface and common project database.[4] Using a common user interface allows for easy transition from one tool to another thereby not only increasing productivity but also saving dollars in development costs of tools. These enhancements are necessary to make the tools robust and user-friendly. If these tools are not developed to high quality standards it will make the task of selling quality engineering in system development a difficult task.

3.0 Assistant for Specifying the Quality of Software (ASQS)

Todays software produced for DoD systems must demonstrate a variety of software quality factors, such as reliability, maintainability, portability, and flexibility, as well as cost-effectiveness. Meeting these goals requires rigorous techniques for selecting appropriate quality factors, balancing quality levels and cost, specifying goals, and evaluating achieved quality. Based on RADC's framework for software quality, acquisition managers are provided with a methodology to consider, balance, specify and evaluate software quality requirements.

The Specification Guidebook requires a substantial time investment to learn and to apply. Applying the Specification Guidebook effectively requires knowledge of software quality concepts and methods, in addition to knowledge of the mission area and system specifics. Individuals who are responsible for high-level management and quality issues often do not have time to familiarize themselves with the more technical aspects of quality. Also, significant experience applying the Specification Guidebook is required before it can be used effectively. ASQS addresses these issues and aids in the specification of software quality requirements.

The objective of ASQS is to make the Software Quality Specification Methodology accessible to DoD acquisition managers. Elements of this objective can be summarized as follows:

- Assist acquisition managers in selecting appropriate software quality factors and in balancing quality levels and cost tradeoffs. Interact with the acquisition manager to translate their perception of the system characteristics and needs into required software quality factors.

- Assist acquisition managers in selecting the required software characteristics needed to achieve the software quality factor goals. Interact with the acquisition manager to translate their perception of the system characteristics and needs into required software characteristics (criteria, metrics, metrics-elements, and weightings).

- Assist acquisition managers in determining the potential effect of different system characteristics and development approaches on software quality factor goals. For example, determine the effect of a particular DoD-STD-2167A tailoring on the expected ability to measure required quality attributes.

• Assist acquisition managers in assessing compliance of software with the specified quality goals.

ASQS automates the software quality knowledge, specification methods, and the process of relating these to the mission area and system-specific knowledge. This allows users to specify software quality based on their knowledge of the mission area and system characteristics, without requiring them to develop and apply expert skills in software quality concepts and methods. ASQS is an expert system in the sense that it provides the expertise of a software quality expert.

ASQS allows users to develop a first-level quality specification in a short period of time and refine the specification throughout the program's life as quality issues evolve. It permits specification prior to full-scale development, including concept exploration, and demonstration and validation. Also, ASQS can be used to specify quality goals during post deployment. It is an expert tool which allows quality specification to play a major role in all phases of system development.

ASQS allows users to tailor the framework to their program without becoming embroiled in metric-level details. It plays the role of an expert who translates the user's high-level system characteristics and needs into low-level software characteristic requirements and presents the results in the user's high-level system requirements language.

ASQS secondary objectives are: 1) to capture the history of selections, decisions and their rationale provided by software quality specifiers, and 2) to provide an environment for viewing observed quality rates. The elements of these objectives can be summarized as follows (see FIGURE 3):

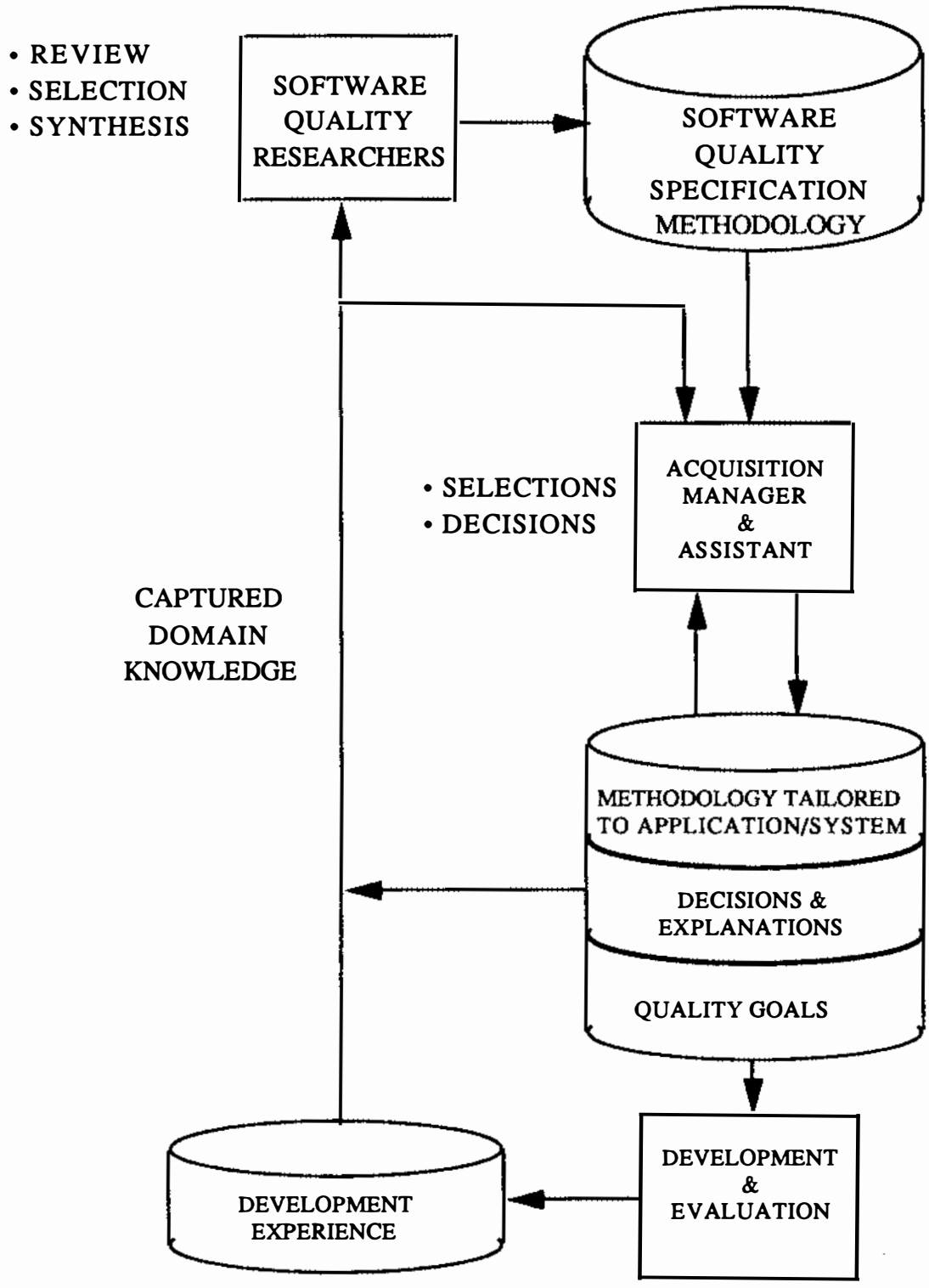
• Capture selections, decisions, rationale, and the outcome of decisions to contribute to the rationale for the software quality specification at a given stage in the project's life cycle.

• Capture selections, decisions, rationale, and the outcome of decisions to contribute to specifications for future projects.

• Capture selections, decisions, rationale, and the outcome of decisions to contribute to software quality research and enhancement of the specification methodology.

• Allow users to view observed quality rates and their relationship to specified and achieved quality for programs with characteristics similar to their own.

ASQS provides for the capture of domain knowledge about software quality concerns which spans the range of DoD applications and systems. The captured knowledge resulting from the application of ASQS can be reviewed by software quality researchers for incorporation into the baseline specification methodology.



THE ASSISTANT PROVIDES A MECHANISM FOR CAPTURING KNOWLEDGE ABOUT SOFTWARE QUALITY.

FIGURE 3

In short, ASQS is designed to provide automated support during every phase of the software development process. ASQS has the following four features. First, its user-friendly Consultation Session bridges the gaps between software-quality concepts and system needs, and between quality specific terminology and terminology understood by application-oriented software professionals such as acquisition managers. By aiding in defining quality requirements prior to full scale development, ASQS has the potential for saving time and money over the course of a project.

Second, as a project historian ASQS assists project managers and programmers throughout the development cycle. Its documentation and report generating functions help keep everyone on the project up-to-date on the latest version and the rationale behind decisions to maintain or change certain features. It is also possible to trace versions back to inception.

Third, ASQS assists software evaluation at every phase. It puts the output of software measurement tools in a meaningful framework, comparing the calculated scores against the target goals based on the defined system requirements.

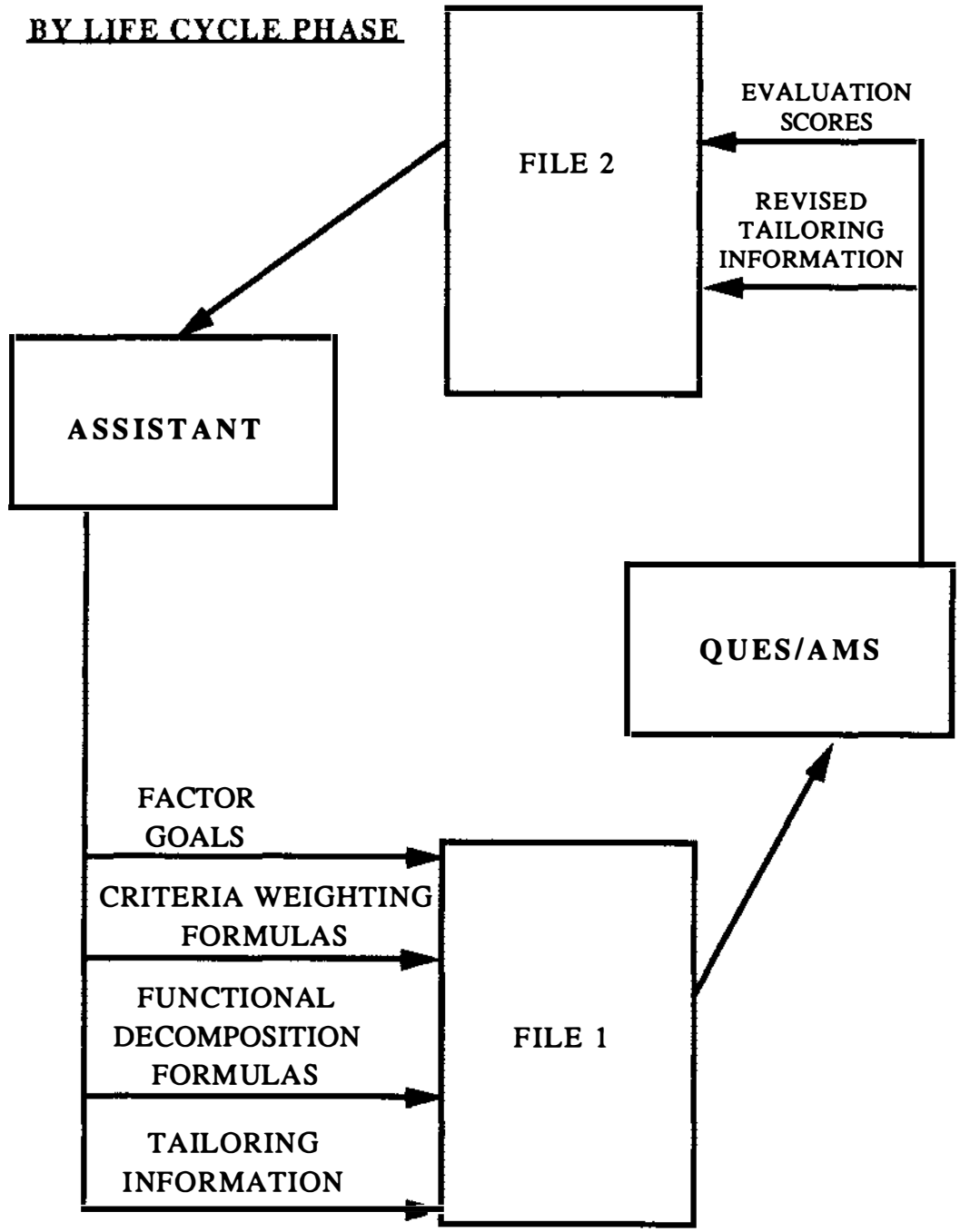
Fourth, ASQS assesses the quality needs and characteristics of the project. It provides guidance for improving the consistency and feasibility of obtaining the software quality goals. It can tell the user where the problems and shortcomings exist and suggest modifications to achieve more desirable and/or realistic goals.

ASQS is an expert system designed for users with a wide range of abilities. For those who are less familiar with the intricacies of software coding, ASQS can translate a general concept of system functional requirements into a framework of quality metrics. Yet even for those who already have years of experience in quality metrics, ASQS can eliminate much of the tedious and cumbersome processes of current methodologies.

The current version of ASQS is a feasibility model created to demonstrate the concept of automated assistance for software quality specification. It contains the essential tool components for ranking software quality factors, considering factor interrelationships, tailoring the framework, and developing numerical goals. It also contains over 100 rules which illustrate these concepts. The current rules support the development of an initial quality specification for the 13 software factors based on system quality factors. Most of the detailed rules are in the area of reliability.

More work is required on ASQS in order to develop a mature system. Further development involves the creation of more rules: to prioritize each of the 13 factors, to characterize interrelationships, to determine the applicability of metrics, and to establish numerical goals. Also, rules will be added first to complete a generic system for one mission area. A generic system allows a user in a specific mission area to have a starting point for system decomposition and specification. The user can copy part or all of the generic case model for his project. An interface to QUES is necessary so that the

specified goals and tailored framework from ASQS can be sent to the evaluation tool, and then the information sent back to ASQS for assessment of compliance (FIGURE 4). Since QUES is replacing the AMS, data currently in AMS can be transferred from the AMS to the QUES only. There will be no data path from QUES to the AMS. Finally, additional tool components to address relative cost tradeoffs, comparison of goals to observed project data, detailed assessment of compliance and hard copy reports are envisioned.



INTERFACE BETWEEN THE ASSISTANT AND QUES/AMS
FIGURE 4

4.0 Automated Measurement System (AMS)

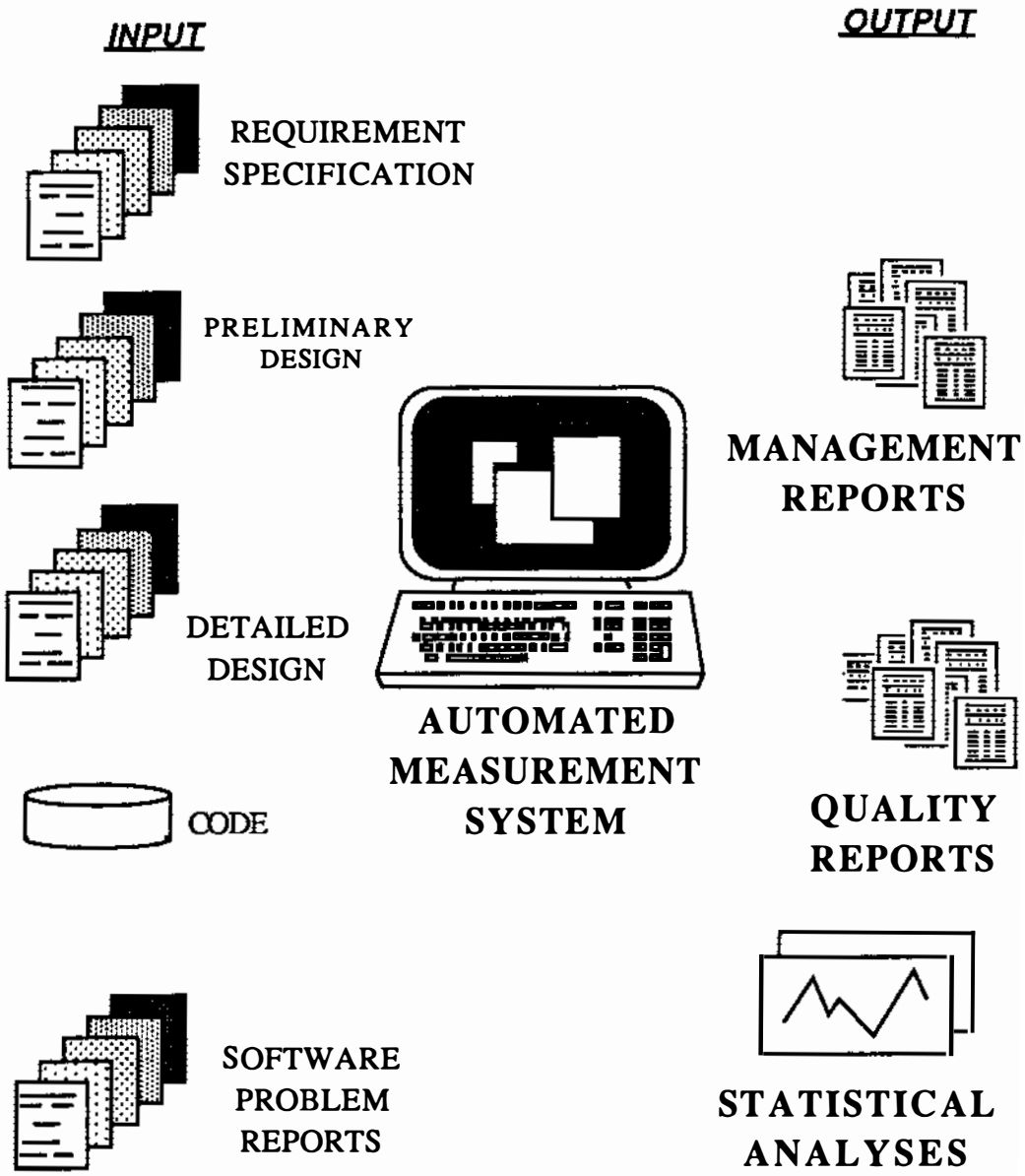
There are several problems with applying the software quality evaluation part of the software quality methodology. Manual data collection is error-prone, inconsistent, time consuming, and expensive. The technology has not been validated and project managers for the most part are concerned about cost and schedule in managing their projects. Measurements have not been proven because data collection and analysis is too expensive. In trying to alleviate the problem of data collection, RADC has developed a tool to automate the data collection and analysis of software measures. Using this tool will reduce the cost of data collection and analysis; thereby, making quality evaluation measurement palatable to software developers and researchers. As more empirical results become available, statistical analysis can be performed to validate the framework and establish baselines for new software development projects.

The tool that was developed to automate the Evaluation Guidebook is called the Automated Measurement System (AMS). This tool will lessen the cost of data collection and analysis and will provide a vehicle for transitioning the technology to others concerned with developing quality software. The objective of the AMS is to support the RADC measurement framework by providing a capability to collect data at all phases of the software life cycle and provide acquisition managers with a means to quantitatively specify quality goals and a capability to measure progress towards those goals (FIGURE 5). In one scenario, the specification of goals will be done by ASQS and sent to AMS. In another scenario, ASQS is not available, and therefore AMS must be able to accept manual specification of quality goals so that the products can be evaluated. The validity of the AMS assessments and predictions depend upon several factors:

- the reliability of the framework (i.e., the questions and the combining equations for deriving metrics, criteria, factors)
- a properly tailored implementation of the RADC framework
- the reasonableness of the data manually supplied by the AMS users

In keeping with the framework, AMS provides a vehicle for taking quantitative measures and measures of attributes to a level of software oriented attributes which provide the required quality and finally to a level that a user understands as the quality of the product.

A highly flexible tool was selected as the number one priority since the measurement framework is subject to revision. For example, revision could be made due to standard changes (i.e., DoD-STD-2167 and DoD-STD-2167A) and technology changes (i.e., Ada, Object-Oriented Design and high performance architectures). Any future changes could be made with very little impact to the tool because additional functionality to suit the new requirements could be added.



CONCEPT OF AUTOMATED MEASUREMENT TOOL

FIGURE 5

The AMS collects, stores and analyzes software measurement data for use by software acquisition and software project managers. The underlying philosophy of the AMS is based on a framework consisting of a set of 13 software factors (i.e., reliability, maintainability, reusability, portability, interoperability, usability, integrity flexibility, expandability, verifiability, correctness, survivability, and efficiency) which are associated with high level concerns of software quality.[5] It provides the capabilities to monitor the overall software quality of a project. Also, it stores information on the status of software problem reports. This information enables a user to analyze the quality data in terms of fault density based on the problem reports.

Data is input both automatically and manually to the AMS. Automated collection during the requirements and design phases is provided through an indirect interfacing with off-the-shelf requirements and design tools. The tools currently interfaced to the AMS are the Requirements Specification Language/Requirements Engineering Validation System (RSL/REVS) and the Software Design and Documentation Language (SDDL). Automated collection is provided for both the Fortran and Ada programming languages during the coding and testing phase of Full Scale Development of the software life cycle. Approximately ten percent of the requirements data, forty-six percent of the design data, and sixty percent of the implementation data are automatically collected by the AMS. Collection of data is accomplished by a software analyzer that interfaces with the source code (RSL/SREM, SDDL, and Fortran or Ada). Data collected by the Automatic Data Collection Function (ADCF) is translated to answers for the software evaluation Data Collection Forms (DCFs) by the Calculate Metrics Function (CMF). These calculated measures are subsequently stored in the AMS database for future use by the Database Management Function (DBMF). Data that can not be collected automatically is entered manually and is controlled by a general forms manager. The AMS can be easily enhanced to interface with other requirements and design tools as well as process other programming languages.[6]

The CMF takes values from the software evaluation DCFs and calculates values for the RADC quality framework. The metric elements are combined to form metrics, the metrics are combined to form criteria, and the criteria are then combined to form each of the thirteen quality factors. All intermediate calculations are retained in the AMS database, thereby allowing comparisons between actual and specified values. Retention of the intermediate calculations also allows developers to observe changes in software quality across several development phases. The CMF is totally data-driven. The equations used to calculate the metrics, criteria, and factors are stored in the AMS database. This allows the AMS equations to be easily adapted to support changes in the quality framework.

The software quality data collected and stored by the AMS is analyzed and displayed to the user via the AMS Report Generator Function (RGF). Reports are keyed to a variety of users (programmers, project managers, acquisition managers). Reports can be generated textually as well as graphically. An example of such a report are the bar graphs which identifies all CSCI's or units that do not meet the

"goal" values for the factor Flexibility (FIGURE 6). Reports of this type can be generated for each of the thirteen quality factors. Another example of a report is the goal comparison display report which includes all factors for a unit as shown in FIGURE 7. Other reports which can be generated include a report on historical trends of quality factors which provides a view of quality across the life cycle phases and complexity reports (i.e., McCabe's Cyclomatic and Halstead's) which can be used to determine the units that need to be reviewed and recoded because they are too complex and may affect the overall quality of the software system.

The AMS collects and manages a large amount of measurement data. The thrust of the AMS is to significantly reduce the cost of applying quality measurement to a project and increase the inherent quality of the developed Mission Critical Computer Resources (MCCR) system. One way to increase reliability is to keep the complexity of units to a minimum since there is a correlation between complexity and number of errors that a unit will have. Application of the framework will provide immediate feedback regarding the overall quality of the software as it is being developed. This will eliminate rework of software and increase overall productivity. Quality/productivity benefits include:

- consistency of collected data via the AMS automated data collection mechanisms
- interactive forms manager for manual data collection
- automated data reduction which provides error-free calculations and removes software development personnel from the tedious and error-prone task
- timely information which is made available for virtually "on the spot" quality analysis
- potential software quality problems identified via the AMS report subsystem which provide managers the choice of "correcting" the deficiency immediately or delaying until later in the development life cycle

In addition to the standalone version, the AMS is integrated with Software Life Cycle Support Environment (SLCSE). AMS can interact with data from the SLCSE database and perform measurement analysis. Using this type of tool interfacing could potentially increase the amount of data which can be automatically accessed.

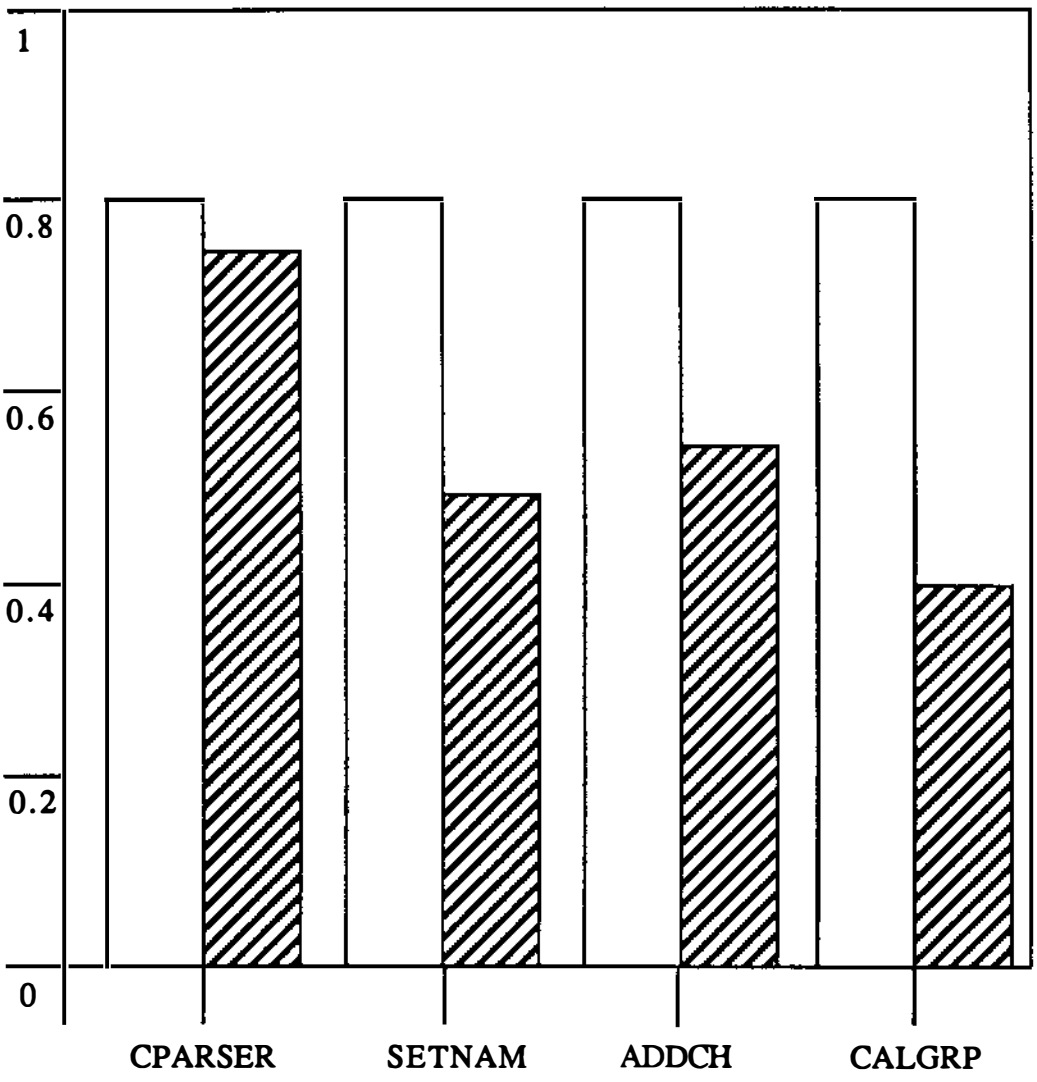
5.0 QUality Evaluation System (QUES)

To tackle the quality problem at all phases of the life cycle and provide quality results in a real-time manner, RADC awarded a contract to develop a quality analysis tool in December 1987. The objective of the QUality Evaluation System (QUES) is to integrate and extend the results of RADC's long-standing software quality program to:

QUALITY FACTOR EXCEPTION REPORT

PROJECT: AMSF
PHASE: E
DATE: 1 JUN 89

FACTOR: FLEXIBILITY
LEVEL: UNIT



UNITS BELOW GOAL VALUE

GOAL ACTUAL

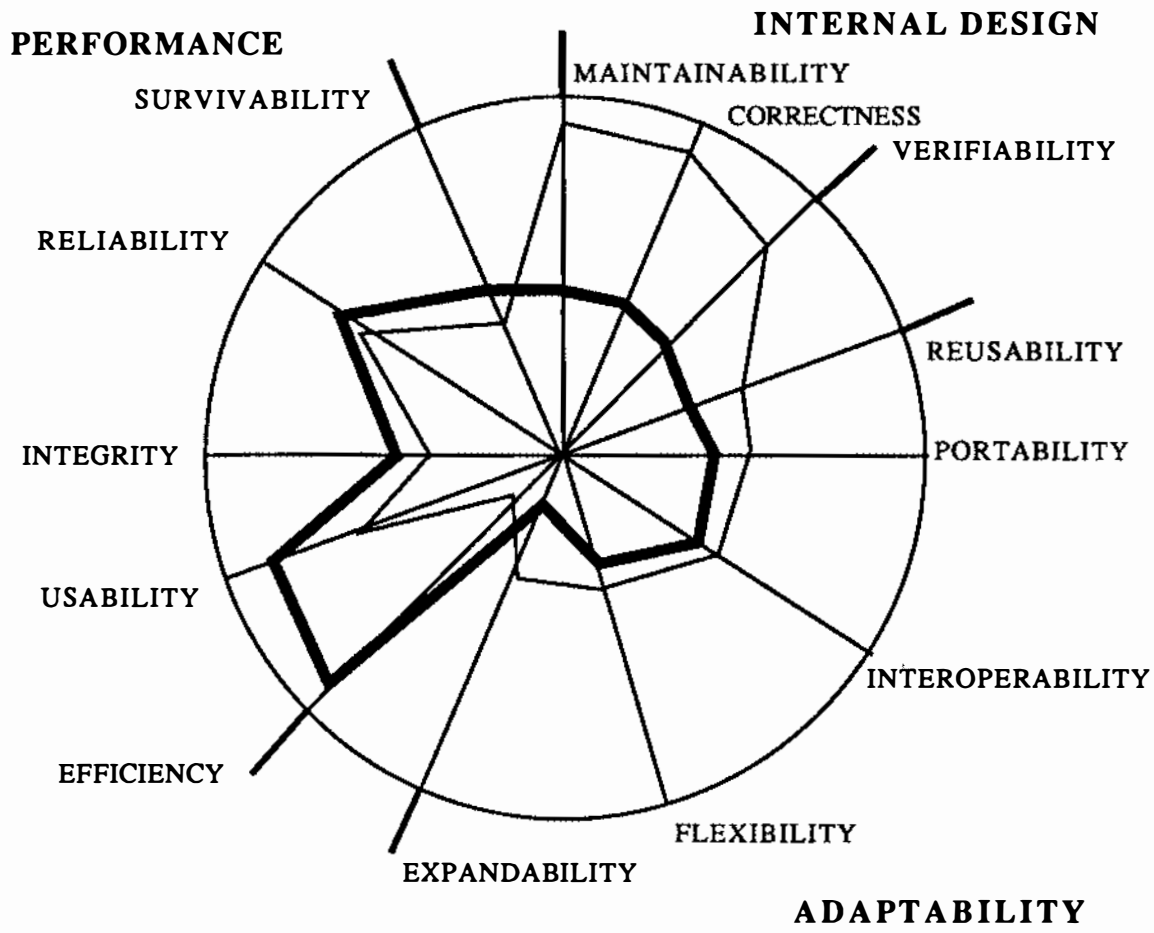
EXAMPLE QUALITY FACTOR EXCEPTION REPORT
BAR FORMAT

FIGURE 6

QUALITY FACTOR GOAL REPORT

PROJECT: AMSF
ENTITY: AMSSYS

LEVEL: UNIT
DATE: 1 JUN 89



EXAMPLE GOALS REPORT
KIVIAT FORMAT

FIGURE 7

• Provide a usable and flexible automated QUES to facilitate automatic data collection and analysis as an aid to improving the quality of software products and the productivity of requirements engineers, designers, programmers, test personnel and test managers who develop the products. This effort will significantly enhance the AMS to provide automated support both in a standalone mode and as an integrated tool in the various configurations of the Software Life Cycle Support Environment (SLCSE).

• Augment the existing software measurement and assessment framework based on application feedback and on automatability studies. The enhanced quality engineering approach provided by QUES will define the next generation of software quality technology supporting Air Force mission-critical software.

• Apply the software quality framework to the development of QUES to gain additional experience with quality engineering methodology and to guarantee that the quality goals of QUES are met. The existing AMS will be used to support this activity.

QUES, with its integral set of software quality engineering methods and tools, provides a cohesive environment that supports a broad range of engineering management and quality assurance roles.

QUES in a standalone mode, provides an Interactive Quality Engineering capability that supports the quality engineering process throughout the life cycle. If ASQS is available, then the specification of quality goals would be done automatically. However, if ASQS is not available QUES must be able to stand alone and contain some of the capabilities of ASQS. QUES provides facilities to:

• Specify quality factor goals and allocate these goals to the specific software components using a spreadsheet style interface.

• Enter metrics data via data collection forms.

• Request automated collection of metrics data.

• Input to project-specific checklists.

• Display and report on quality goals, measurements and trends using a variety of textual and graphical representations:

  ◊ Computer-generated documentation
  ◊ Tables
  ◊ Line graphs
  ◊ Pie charts
  ◊ Kiviat graphs
  ◊ Hierarchy graphs
  ◊ Density diagrams
  ◊ Animated trend displays

QUES provides an automated measurement capability for automated

collection of metrics data from work products developed on the project. Within the context of a life cycle software engineering environment, such as SLCSE, it is possible to acquire most of the quality measurement data without reliance on manual inputs to data collection forms. A robust set of language and database filters allow metrics collection on the full range of work products, from requirements through code. QUES's metrics daemon provides continuous automated measurement, providing current status on the data collection forms throughout a project development.[7]

These capabilities support the full spectrum of engineering management and assurance roles:

- Acquisition Manager: The acquisition manager uses QUES to specify and measure quality goals for specific programs. QUES assists the acquisition manager in prioritizing quality goals for the program's application.

- System Engineer: During the system concept and system/software requirements analysis phases, system engineers use QUES to specify and allocate quality goals and weights to the system and CSCIs.

- Quality Expert (e.g., reliability engineers, performance modelers, and human factors engineers): Quality experts use QUES to fine-tune or enhance data collection to support various measures of a particular quality.

- Software Requirements Engineer: During the system/software requirements analysis and software requirements analysis phases, software requirements engineers use QUES to specify and allocate criteria and weights to CSCIs and lower level components (CSCs, and CSUs).

- Project Manager: QUES assists project managers in monitoring the overall progress of the project with regard to quality goal achievement. Project managers can choose from a variety of QUES reports to pinpoint problem areas and identify corrective action.

- Development Engineer: Software designers, programmers, and test engineers use QUES to obtain measurement reports and assists development engineers in managing their work assignments by providing checklist reminders, assessing goal achievement, and highlighting problem areas.

- Quality Assurance Engineer: Quality assurance engineers use the reporting facilities of QUES to assess goal achievement and to alert management to discrepancies. Quality assurance personnel provide independent inputs to QUES to calibrate and verify project measurements.

- IV&V Personnel: IV&V personnel use QUES in much the same manner as quality assurance engineers and can also use QUES reporting facilities to compare data across projects in support of broad acquisition management concerns.

QUES provides a set of framework management facilities to support Measurement Technology Experts. These facilities are used during the proposed development effort to refine and update the software quality engineering framework based on application experience. In the future, experts in measurement technology will be able to use QUES to add new quality factors, criteria, and metrics; and to change the standard framework to accommodate new technology, standards, and results of validation programs.

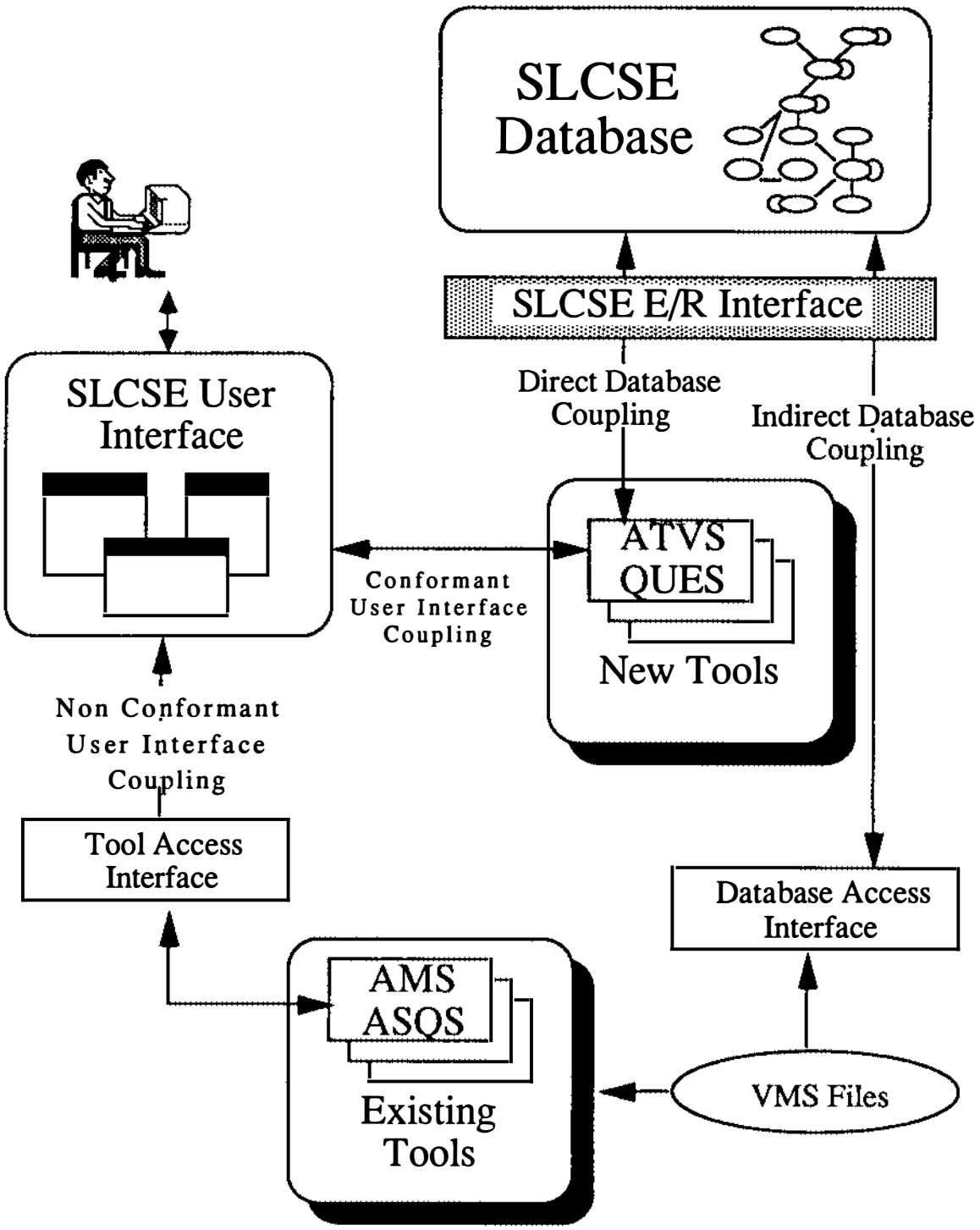
QUES provides project quality management utilities to support Project Methodologists. Project methodologists use QUES to tailor the framework according to specific project attributes such as application domain or documentation requirements and implementation language. Project methodologists also use QUES to establish and review phase completion checklists and to schedule project-level data collection.

Utilities are available for QUES administration (performed by Environment Administrators). Environment administrators insure that QUES is fully integrated with the host development environment (e.g. SLCSE), that all QUES users have appropriate levels of access to necessary data, and that capabilities for automated data collection are installed and scheduled as required.

Because of its availability in both standalone and SLCSE configurations, QUES can be widely distributed, providing a valuable tool for improving the quality of mission-critical software.

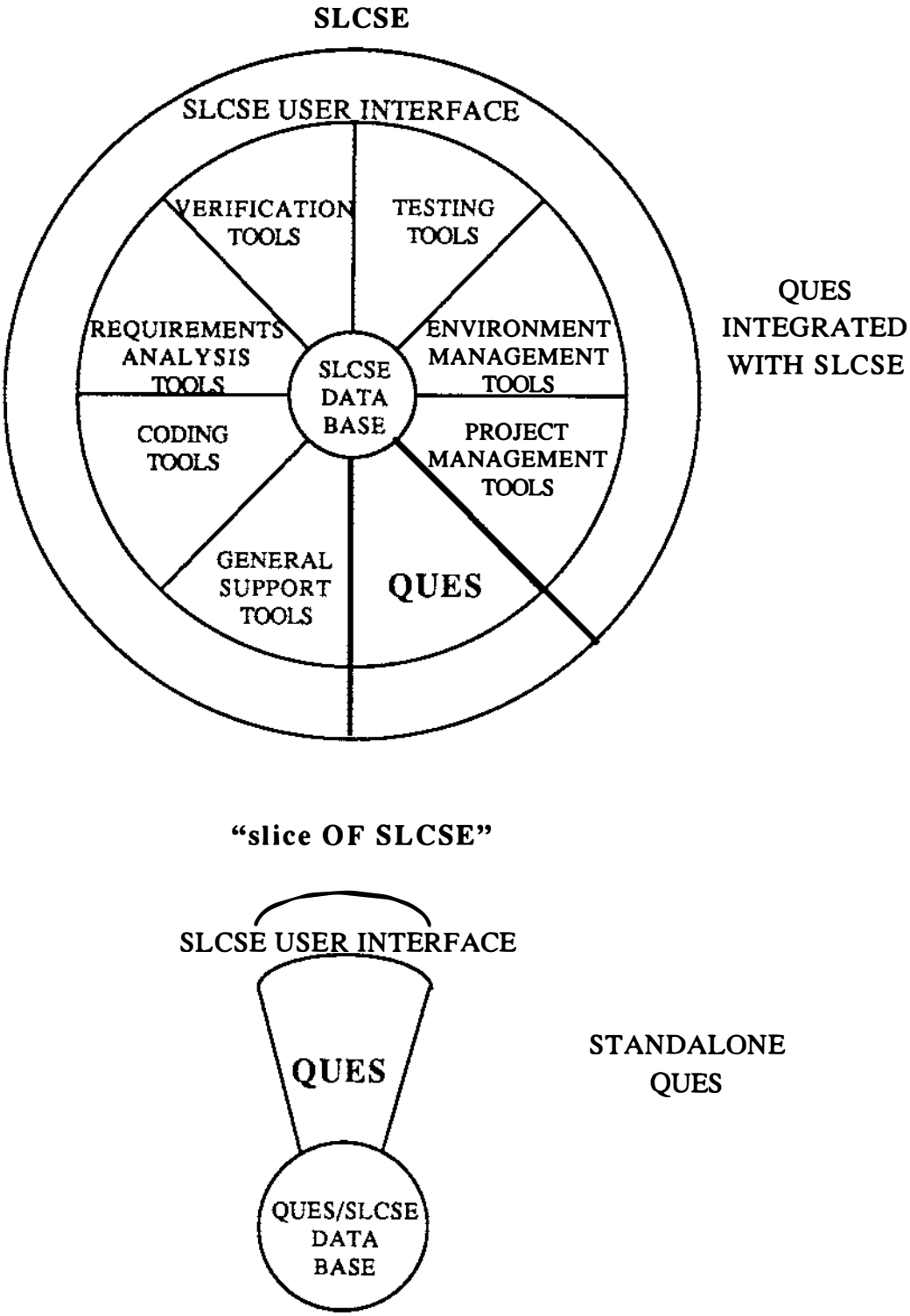
The approach to the development of QUES is the concept "slice of SLCSE". The SLCSE infrastructure is composed of the user interface subsystem and the database subsystem. A top level view of the SLCSE architecture is shown in FIGURE 8. Rather than developing a semi-loosely coupled version and a standalone version of QUES which are two totally different versions, the "slice of SLCSE" (FIGURE 9) approach allows the differences between the two versions to be reduced significantly. The Ada Test and Verification System (ATVS) will be coupled with QUES providing the data collection mechanisms to automatically collect Ada metrics. ATVS will collect Ada metrics from detailed design, coding and testing phases. QUES will be integrated with the SLCSE infrastructure in both the SLCSE version and in the standalone version. The "slice of SLCSE" approach results in a tightly integrated tool.[8] Being tightly integrated has the following advantages:

- same database manager is used for the QUES schema and the SLCSE schema
- same user interface manager is used for both QUES and SLCSE
- additional user interface and database utilities developed for QUES will be available to other SLCSE tools
- data integrity and consistency is easier to maintain when all data is in the same database



SOFTWARE LIFE CYCLE SUPPORT ENVIRONMENT (SLCSE)
DESIGN

FIGURE 8



"slice of SLCSE"
ARCHITECTURAL APPROACH
FOR QUES

FIGURE 9

The QUES architecture provides a well-defined interface (both procedural and database model) whereby appropriate language filters are configured to support automated measurement of language-based work products. The current AMS provides an effective Fortran filter that will be incorporated into the QUES. AMS also provides RSL and SDDL filters that will be included in QUES. Other requirements and design languages (e.g., PSA/PSL and Ada PDL) could also be incorporated. An internal Ada parsing engine from the ATVS will be included to provide a sophisticated Ada filter that meets the long term needs of QUES and provides the necessary metrics for Ada during design, coding and testing.

The database method that QUES will use is the Entity-Relation (ER) database. This database schema has been adopted by SLCSE. The QUES database approach reflects further demonstration of the innovative yet pragmatic approach for large-scale software engineering environment databases, as follows:

- By integrating the QUES schema with that of SLCSE and by using the SLCSE ER database interface, the approach provides significant flexibility to 1) adapt QUES as SLCSE is adapted for different installations, 2) tailor the QUES for different projects and 3) provide for significant extensibility as SLCSE and quality technologies evolve.

- The potential for growth of the QUES is significant because of the open SLCSE database architecture that the QUES toolset utilizes.

- QUES benefits, in the SLCSE configuration, from performance enhancements and optimization that will be performed on SLCSE. In the standalone configuration, high performance Ada ER Database Management System (DBMS) will provide increased performance.

Because the QUES architecture utilizes the SLCSE architectural approach, it will benefit from future SLCSE extensions to support Knowledge Based Software Assistant (KBSA) tools. The SLCSE, and therefore the QUES, have the following features to support future KBSA integrations:

- The ER semantic interface to information is sufficiently similar to AI-based semantic networks to allow an efficient interface to be constructed so that AI-based applications to access the SLCSE database.

- The SLCSE-maintained "meta-schema" (database containing the ER schema) provides a knowledge base that allows adaptive AI-based algorithms to be supported.

- QUES will be designed to integrate with the LISP-based ASQS.

- The extensive QUES ER database provides the basis for future rule-based methodology and corrective guidance assistance.

- The QUES support of an extensible software quality framework definition with tailorable language filters allows the future addition of factors, criteria and metrics such as those which characterize the software quality of AI-based systems and languages.

In addressing tools and methods, one must be aware of the need to develop quality technology that aids in producing reliable and maintainable mission-critical, embedded systems for the Air Force. The development of this measurement tool will address several issues necessary for this technology to be accepted and used as follows:

- Cost of application versus expected benefits -- If the methodology and tools require too much effort to use, then contractors will choose to use them only when contractually required. The benefit of a quality measurement capability will be accomplished only when it is tightly integrated into the contractor's methods and tools rather than produced simply to meet contractual requirements.

- Effectiveness -- If contractors apply the methods and tools, the information should indicate that they have good quality or bad quality. If the tools and methods are unreliable, then contractors and acquisition managers will be very hesitant to use the methods and tools again.

- Applicability to phases and roles -- Effective methods and tools must apply throughout the life cycle and be applicable to all of the roles performed by personnel involved in the development. The current methods and tools must specifically support the current phase and the current user in a customized fashion.

6.0 Conclusions

The development of a methodology and tools to support the quality engineering process requires commitment and money. Management must be willing to accept the technology and be committed to mandate its use. It is true that the numbers are not validated but the indication of potential problems can not be ignored any more. Using tools to aid in specifying quality requirements and evaluating to see that these requirements are being met should be promoted and encouraged. Quality should be built into the software, rather than tested after the fact to see if it does its intended function or that it is supportable. With the rising costs of developing and supporting systems we have to rethink our philosophy of just cost and schedule during development. The money spent up front to collect and evaluate quality data during full scale development in the requirements, design, coding and testing phases could potentially save millions of dollars in post deployment support where it is costly to correct errors and make changes to get the system to do its intended function. This technology however, will not provide a magical solution but will provide a basis for tracking and making critical decisions in the development of the software system. As Dr Fred Brooks has written there are "No Silver Bullets."[9]

7.0 Acronyms

ADCF	Automatic Data Collection Function
AMS	Automated Measurement System
ASQS	Assistant for Specifying the Quality of Software
ATVS	Ada Test and Verification System
CMF	Calculate Metrics Function
CSCI	Computer Software Configuration Item
CSC	Computer Software Component
CSU	Computer Software Unit
DBMF	Database Management Function
DCF	Data Collection Form
DoD	Department of Defense
IV&V	Independent Validation & Verification
KBSA	Knowledge Base Software Assistant
MCCR	Mission Critical Computer Resource
QUES	QUality Evaluation System
REVS	Requirements Engineering Validation System
RGF	Report Generator Function
RSL	Requirements Specification Language
SDDL	Software Document Design Language
SLCSE	Software Life Cycle Support Environment

8.0 Bibliography

[1] T.P. Bowen, G.B. Wigle, J.T. Tsai, "Specification of Software Quality Attributes" Technical Report, Boeing Aerospace Company for Rome Air Development Center, RADC-TR-85-37, 3 Vols, February 1985 (NTIS AD-A153988, AD-153989, and AD-A153990).

[2] L. Kahn, S. Keller, "Operational Concept Document for the Assistant for Specifying the Quality of Software (ASQS)," Dynamics Research Corporation for Rome Air Development Center, December 1986.

[3] T.A. Babst, P.T. Siefert, P.L. Koltun, "Automated Measurement System Program Specification," Harris Corporation for Rome Air Development Center, January 1986.

[4] General Research Corporation, "Software Life Cycle Support Environment - Software Detailed Design Document," Sep 1987.

[5] T.A. Babst, P.T. Siefert, P.L. Koltun, "Automated Measurement System (AMS) Program Specification," Harris Corporation for Rome Air Development Center, June 1987.

[6] T.A. Babst, P.T. Siefert, "Final Technical Report for the Automated Measurement System (AMS)," Harris Corporation for Rome Air Development Center, June 1987.

[7] P. Dyson, B. Bach, Technical Proposal for the "QUality Evaluation System (QUES)," Software Productivity Solutions, May 1987.

[8] W. Wisehart, et al., "C3I Support Environment Definition" Technical Report, General Research Corporation for Rome Air Development Center, RADC-TR-86-54, September 1985.

[9] F.P. Brooks Jr, "No Silver Bullets," COMPUTER, April 1987.

Software Project Assurance

at the Jet Propulsion Laboratory

by

Richard E. Fairley
Distinguished Visiting Scientist
Jet Propulsion Laboratory
and
Professor of Information Technology
George Mason University

Marilyn Bush
Manager, Software Product Assurance
Jet Propulsion Laboratory

ABSTRACT

The Jet Propulsion Laboratory, a division of the California Institute of Technology, is responsible to NASA for conducting scientific investigations of the solar system using automated spacecraft. In recognition of the growing dependence of JPL missions on computer software, a new Software Product Assurance organization has been created. This paper describes the SPA organization and the novel approach to assuring safety, reliability and efficiency of software that is being pursued by the SPA group. Activities, results to date, and lessons learned are presented.

Author's Biosketches

Richard E. Fairley is a Professor of Information Technology at George Mason University in Fairfax, Virginia and a Distinguished Visiting Scientist at the Jet Propulsion Laboratory in Pasadena, California. In addition, Dr. Fairley is Director of the Masters' program in software engineering and Director of the Center for Software Systems Engineering at GMU. He was program chair for the 11th International Conference on Software Engineering and guest editor for a recent issue of the IEEE Transactions on Software Engineering on software engineering education. Dr. Fairley is author of the textbook "Software Engineering Concepts" (McGraw-Hill, 1985). He has a Ph.D. in computer science from the University of California, Los Angeles.

Marilyn Bush currently manages the Jet Propulsion Laboratory's Software Product Assurance Section. She has managed teams responsible for systems engineering and software engineering tasks on projects as diverse as the Deep Space Network and the FAA's Central Weather Processor. Ms. Bush chairs the HQ NASA Software Management Assurance Program (SMAP) Committee on Software Education and serves as the JPL representative to the SMAP Steering Committee. Before she came to JPL, Bush worked at the Transportation Systems Center, Cambridge, Massachusetts; was a tutor at Winthrop House, Harvard University; and founded the Greater Boston Women's Transportation Group and the Boston Professional Council. She is a member of the IEEE Education Committee and holds a B.S. in Electrical Engineering from the University of Pennsylvania's Moore School of Electrical Engineering.

I. INTRODUCTION

The JPL Charter

The Jet Propulsion Laboratory (JPL) has primary responsibility within the National Aeronautics and Space Administration (NASA) to conduct investigations of the solar system using automated spacecraft. JPL is a government-owned Federally Funded Research and Development Center operated by the California Institute of Technology. JPL is an organization of approximately 5000 employees, 3800 of whom are technical. Approximately 2000 have software related jobs.

Missions conducted by JPL include Explorer 1, the first U.S. satellite; the Ranger lunar photo-reconnaissance missions; the Surveyor lunar landers; the Mariner series that explored Mars, Venus, and Mercury; the Viking Mars orbiters; and the Voyager missions to Jupiter, Saturn, and Uranus. Magellan, launched from the space shuttle Atlantis on May 4, 1989 will map Venus from orbit with high-resolution imaging radar beginning in September, 1990.

JPL provides support to NASA for a wide variety of other activities, including the design and operation of the Deep Space Network of global tracking stations. JPL also has a major role in the NASA Earth Observation Systems (EOS) mission. More than 75% of JPL resources are devoted to NASA activities. The remainder of JPL activities involve applied research and development for sponsors such as the Department of Defense, the Department of Energy, and the Department of Transportation.

The Growth of Software on JPL Missions

Although JPL missions have relied on computers and software for operational support for over 20 years, the amount and importance of software has dramatically increased over the last five years. For example, the Voyager spacecraft (1977) made use of approximately 10,000 lines of code for flight software and 1.5 million lines for ground software. The planned Craft-Cassini mission to explore the outer planetary system is estimating 50,000 lines of on-board, flight critical software. Early estimates for the planned EOS project are around 10 million lines of code. In addition, the non-NASA projects at JPL have become increasingly software intensive with passing time.

The Genesis of Software Product Assurance at JPL

By the mid-1980s it became apparent that JPL needed a more disciplined approach to software development. In 1985 the Deputy Director of JPL chartered a "Software-Intensive Systems Study (SISS)." The Study concluded that JPL needed to do a better and a more consistent job of software development, and above all, to learn to control changes in software. As a result of this study, the Systems, Software, and Operations Resource Center (SSORCE) was created in July, 1986. This umbrella organization provides partial funding to five discipline centers (Systems Engineering, Test Engineering, Software Product Assurance, Software Engineering, and Operations Engineering). The purpose of these centers is to implement the recommendations of the SISS Committee. A major role of SSORCE has been to define JPL standards for software.

In the summer of 1987 the Software Product Assurance Section (SPA) was chartered by the Laboratory Director as a permanent organization at JPL to address issues of quality, productivity, and reliability in all JPL software activities.

The remainder of this paper describes the organizational context of SPA at JPL, outlines the support services provided by SPA, describes some other activities of the SPA organization, presents some lessons learned to date, and discusses likely future directions for SPA at JPL.

II. THE ORGANIZATIONAL CONTEXT OF SPA AT JPL

The JPL Matrix Structure

Like all NASA facilities, JPL relies on a matrix management structure. One dimension of the matrix is a set of Program Offices. At JPL there are four Program Offices: Technology and Applications, Flight Projects, Telecommunications and Data Acquisition, and Space Science and Instruments. The other dimension of the matrix encompasses the Technical Divisions - 8 in all: Systems, Earth and Space Sciences, Telecommunications and Engineering, Electronics and Control, Information Systems and Institutional Computing, Missions Operations, Observational Systems, and Mechanical and Chemical Systems. (The administrative divisions also reside in the matrix but they will not be discussed in this paper.)

In typical matrix fashion, the Program Offices contract with the Technical Divisions to complete project work. If a systems engineer is needed on a flight project, for example, the project manager - who resides in a Program Office - contracts with the Systems Division for the services of that engineer. Figure 1 illustrates the JPL matrix structure.

The SPA Structure

The Software Product Assurance Section resides within the JPL matrix as follows. Functionally, SPA is part of the Reliability and Safety Division, which reports to the Assistant Laboratory Director (ALD) for Engineering and Review, who in turn reports to the Director of the Laboratory. SPA has 105 employees in four groups: Flight Systems, Ground-Based Systems, Technical Applications, and Methodology. SPA personnel are assigned to projects in teams of 1 to 5 members. There is an SPA Manager for each project. An SPA team may consist of both generalists and specialists in areas such as flight, ground, reliability, performance, particular standards, or inspection techniques. SPA team members are usually assigned full time to a project throughout the lifetime of that project. Some SPA members play the roles of facilitators and trainers while other play the more traditional gatekeeper role.

Within the JPL matrix structure SPA, like the Technical Divisions, is commissioned on a case by case basis by projects in the Program Offices to perform certain services for those projects. Project managers are not required to utilize the services of SPA. SPA personnel work with the project manager but retain their independence through the Assistant Laboratory Director for Engineering and Review. When project managers involve SPA in their projects, it is at their option and on the basis of negotiated services for fee to be provided by SPA and paid for from the project manager's budget. Once signed on to a

JPL MATRIX ORGANIZATION

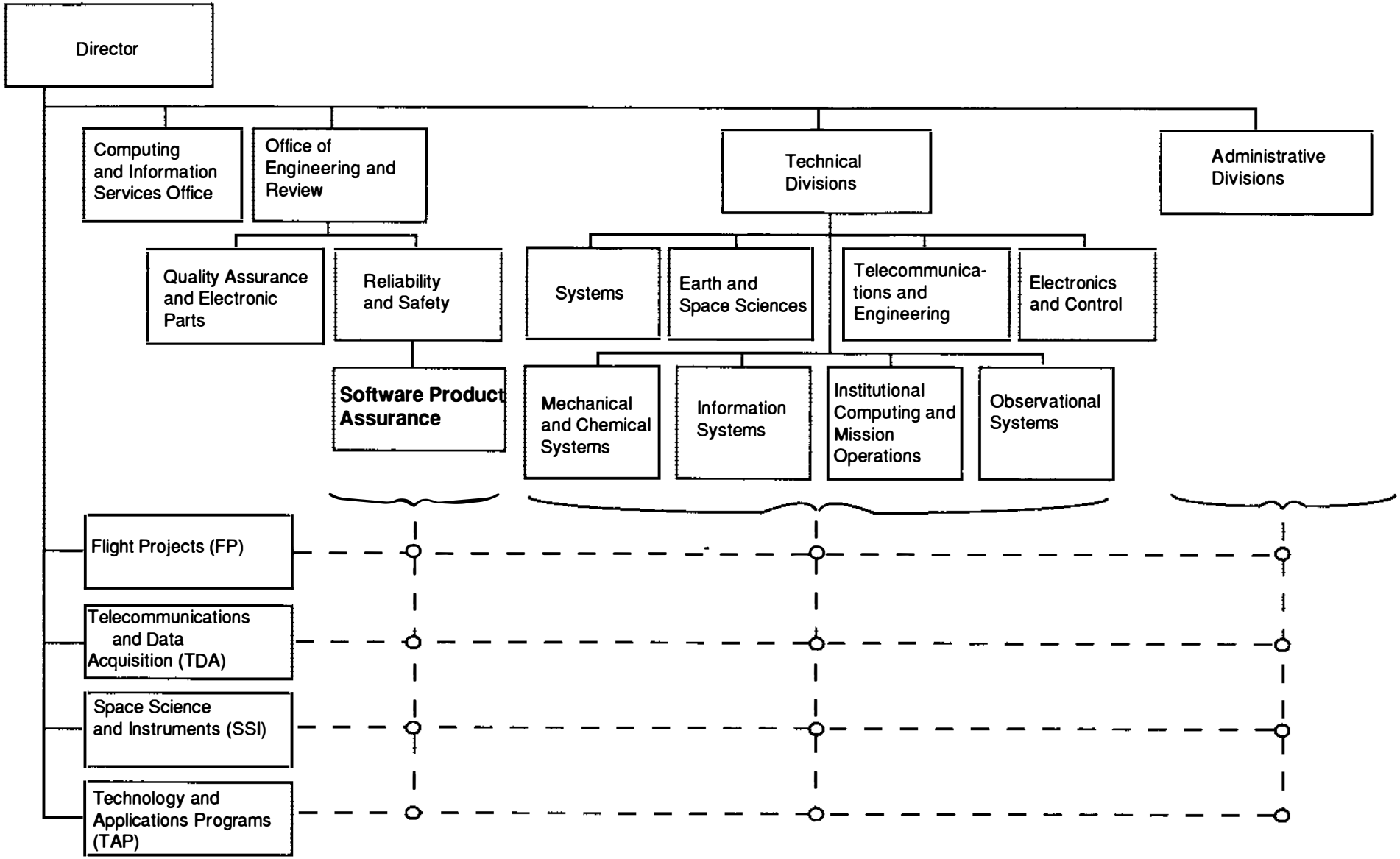


Figure 1.

project, however, SPA personnel retain their independent reporting channel to the ALD for Engineering and Review.

This unusual, if not unique, arrangement has many positive aspects. For example, SPA personnel must justify their existence by collecting and presenting convincing evidence that their efforts represent a value-added contribution to the projects in which they participate. At project initiation, SPA personnel meet with the project manager to describe the services and capabilities they can provide and to indicate the likely costs and benefits to the project of each potential SPA activity. This approach thus results in a metrics focus, because SPA personnel must collect data and demonstrate the benefits of their activities; it results in a focus on high-leverage, high-payoff activities; and it results in tailoring of an individual SPA plan for each project. Some of the problems encountered with this approach are discussed in the Lessons Learned section of this paper.

The following section describes the services available from SPA and presents the format of an SPA plan.

III. PROJECT SUPPORT BY SPA

Services Provided

The novel aspects of SPA at JPL arise from our belief that software product assurance should be involved as a "value-added" partner in every aspect of the development process rather than as a certifier of end-item products. In this regard, we are in agreement with Loy [1], who argues that the traditional autonomous SQA organization should be integrated into all aspects of the development process and that the scope of SQA should be viewed in the broadest of contexts.

To convey this attitude, the name "Software Product Assurance" was chosen at JPL, rather than the more traditional "Software Quality Assurance." The primary role of SPA is to assist the project manager in achieving successful product development. The first tenet of SPA at JPL is that software quality is best achieved through an effective development process. The second tenet is that SPA is chartered to assure an effective process.

Services that can be provided by SPA include evaluation of various documents for format and technical content (the Lifecycle Management Plan; Requirements and Design Documents; the Interface Document; Implementation, Test and Release Document; and Related System and Subsystem Documents); assistance with formal inspections; evaluation of the configuration management process; evaluation of testing requirements, plans, procedures, and results; participation in milestone reviews; assurance of standards compliance; evaluation of metrics efforts; assistance in the SPA aspects of contractor developed software; and assessment of the software process being used on a project [3]. In addition, SPA personnel are available to meet with the project manager at the outset of a project to provide advice on setting up an effective and efficient software development process for the project.

Negotiating and Tailoring an SPA Plan

There are two factors that must be taken into account when considering the role of SPA in a JPL project: first, it is traditional within the JPL culture that the project manager has full responsibility and authority (subject to review, of course) for the system being developed and for all of the work products associated with the system, including the software products. Project managers can choose to involve, or not involve, SPA as they wish, and a project manager can pick and choose among the various services available to a project from SPA. There are no organizational mandates for SPA involvement in projects and no automatic allocation of project funds to SPA, as occurs in many organizations.

The second factor to be considered is that SPA must be careful to allocate limited personnel resources to high leverage, high payoff activities. Taken together, these two factors indicate that an SPA plan must be negotiated and tailored for each project in which SPA becomes involved.

When completed, the SPA plan is considered to be a contract between the SPA manager and the project manager. The SPA manager commits SPA resources to perform certain specific quality assurance activities and the project manager commits a specific level of support for the SPA activities. Major items in the SPA plan should include specification of the SPA tasks to be performed and the criteria for their selection; the SPA procedures and level of SPA support to be provided; and allocation of SPA efforts according to the project tasks, calendar period, and resource level. Once agreed to, the SPA plan should be placed under baseline control and revised as required upon agreement of the SPA manager and the project manager. An annotated outline for an SPA plan is contained in Appendix A.

Recommended Services

Project managers at JPL are typically responsible for planning and developing complex systems that incorporate scientific instruments, computing hardware, software, and telecommunication facilities. Usually JPL project managers are not very knowledgeable concerning software or the software development process. Given that SPA is a service-for-fee organization, a project manager may be uncertain which services will provide the maximum benefit to the project.

These considerations raise the issue of a recommended level of involvement of SPA on JPL projects; in other words, is there a minimum set of activities that SPA should insist on if they are to be effective project agents? For example, should every project have a project library? an automated change control system? inspections? (how many?, what types?), test plans keyed to requirements? traceability matrices?

This issue has turned out to be surprisingly difficult. It is not hard to prescribe a few necessary items, such as those stated above. The difficulty is in drawing the line between required and desirable SPA activities. An approach that is currently being pursued is to extract the procedures, documents, and reviews contained in JPL Standard D-4000: JPL Software Management Standard [2] and examine them for inclusion in the set of recommended activities. This list is displayed in Appendix B of the paper. The list is categorized at levels A, B, C, D, and E, in recognition that different types of projects have

differing SPA requirements. These categories correspond to the JPL mission criticality criteria (A = Critical; E = Low Risk). In general, we have found that the high payoff SPA activities for a software project are: assisting the project manager to formulate a software quality plan at project inception, continuing consultation with the project manager and project team members, use of formal inspections (requirements, design, code and test plan inspections), and flight command assurance.

Lessons Learned

Although SPA has been in existence only a short time, several lessons have been learned concerning the "contracted-service-for-fee" approach to SPA being pursued at JPL. First, and perhaps most obvious, is that limitations on personnel resources dictate that activities and tasks to be pursued be carefully chosen for their likely return on investment.

Second, SPA is most effective when SPA personnel are regarded by the projects they work on as project team members responsible for improving the development process, and thereby, the quality of the resulting work products. On the other hand, SPA must retain organizational independence in order to enforce quality concerns when trade-offs among quality, schedule, and budget are made, or when process issues are slighted by in-house software developers and subcontractors. Development of a collegial atmosphere is of paramount importance, and one of the most difficult aspects of the SPA function. The most effective SPA personnel are those who understand human behavior and are skilled in human relations.

A related lesson learned is a by-product of the matrix structure. SPA, by necessity, must become involved in the work of the Technical Divisions doing the development work. Because SPA is contracted by the project manager to work with the developers, those developers may regard involvement of SPA as lack of confidence by the project manager in their ability to do their jobs. This can result in resentment and friction among SPA personnel, the software developers, and the project manager. An effective approach to overcoming these problems is for SPA personnel to function as facilitators and trainers, rather than as watchdogs.

Still another lesson learned is that SPA personnel must clearly communicate the role of SPA to the project manager during initial negotiations and throughout the lifetime of the project. Some managers have tended to view SPA as a source of additional programmers for their projects. This becomes a critical issue when a project begins to slip the schedule. Because project managers pay for SPA services they may regard it as their right to assign personnel as they see fit. Problems of this nature are not unusual in a matrix structure when the project manager and the functional managers disagree on their respective authorities and responsibilities. On occasion, it is necessary to assign SPA personnel to programming activities because those persons have special skills critical to project success. In these cases, a formal request is made through the Assistant Laboratory Director for Engineering and Review. Upon approval, the requested SPA personnel are temporarily assigned to a Technical Division and new SPA personnel are assigned to the project. In this manner, personnel conflicts are resolved to the benefit of everyone.

IV. OTHER SPA ACTIVITIES

In addition to providing direct support to mission-oriented projects, SPA is involved in several activities that provide institutional support to JPL. These activities include a software metrics program, development of standards, development and offering of training courses, and a research program in the methodology of software product assurance.

Software Metrics

Prior to the formation of SPA, there was no concerted effort to collect or analyze quality and productivity metrics for software development at JPL. But over the last two years, SPA has begun to generate order-of-magnitude data on the productivity and quality of JPL software systems. This work has so far retrieved and reconstructed close to 170 (systems and subsystems) data points representing about half a billion dollars of JPL software projects, over six million source lines of code and over 32,000 person-months of effort over 15 years. This data has been broken down into four areas: flight software, ground software, flight instrument software and Non-NASA project software. Based on the data, SPA is developing four groups of equations with the goal of deriving appropriate cost drivers that can be used to help predict the cost of future software projects [4].

The SPA organization has also projected a list of useful metrics that on-going projects might compile and has provided arguments to justify those metrics. This material includes recommendations about how the data should be presented, where the data will come from within JPL, and how to analyze the data once it has been collected. (One of SPA's services is to help projects analyze data once they collect it.)

Standards Development

A major component of the SSORCE charter that resulted from the Software-Intensive Systems Study conducted in 1985 was to develop a comprehensive set of standards for software development and management at JPL. SPA has contributed to that effort, and continues to do so. In particular, JPL Standard D-4013: Software Product Assurance Standard has been published in draft form and is in revision for publication in final form [5].

Training Courses

One of the most effective ways for SPA to increase the productivity of software engineers at JPL and the quality of the software they develop is through training courses in various aspects of software quality, productivity and product assurance. For example, SPA has developed two courses on Formal (also known as "Fagan") Inspections [6]. One course is for managers, and one is for developers and moderators. A total of about 450 people have been trained by SPA on Formal Inspections in 15 months. More significantly, 176 inspections (normally two hours long or less) have been conducted since March 1988, under the auspices of eight projects (five additional projects are seriously considering them). On the average, JPL is finding five major defects and 13 minor defects per inspection. The average estimated savings to JPL for each inspection is around $200,000.

Other SPA courses in development (for JPL and for NASA) include: Software Risk Management, Controlling Projects through Metrics, Improving Software Productivity, Project Assessment for Managers, and Software Cost Estimating. Along with course development, SPA also produces practical guidebooks on how to improve the software process. Draft versions of a Requirements Handbook and a Tracking and Project Assessment Guide have been developed during the past year.

In addition to courses intended for JPL and NASA, SPA presents workshops to train its own personnel. For example, a one-day software product assurance course is offered and a series of half-day workshops on topics such as "What is quality software? - How to define it and measure it," "A project manager's view of SPA," and "Software Project Risk Management" are offered on a continuing basis.

SPA Methodology Research

Research activities in software product assurance are intended to explore issues and provide recommended approaches that will allow SPA to improve current practices and be prepared to respond to new product assurance requirements in the future. Topics under investigation include the role of product assurance under new development paradigms, such as object-oriented development and evolutionary development; recommended metrics to support cost estimation, status tracking, risk management, and contractor monitoring; quantitative models of software reliability; tools and techniques for performance evaluation; and automated support for the inspection process.

V. FUTURE DIRECTIONS FOR SPA

Given the broad charter of SPA at JPL, many types of activities can be justified under the SPA banner. However, resource and time constraints dictate that new activities be carefully chosen to maximize the cost/benefit of those efforts. Some of the planned and present activities include work in requirements methodology, reliability and performance, defect prevention, cost estimation, risk assessment, and organizational capability assessment. For example, NASA is funding risk assessment and organizational capability assessment projects at JPL.

VI. SUMMARY AND CONCLUSIONS

Software product assurance at JPL is a young and growing organization. During the past two years, SPA has quadrupled both its budget and number of personnel. There are still many avenues to be explored and many procedures to be developed. However, it is clear that SPA has made significant contributions and that the importance of the SPA function will grow with increasing size, complexity, and criticality of software on JPL missions.

References

[1] P.H. Loy, "Enlarging the Scope of SQA," ACM Software Engineering Notes, January, 1989. Also appeared in the 1988 Pacific Northwest Software Quality Conference.

[2] JPL D-4000: Software Management Standards Package, Version 3.0, December, 1988. Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California.

[3] Fairley, R.E., Software Engineering Concepts, McGraw-Hill, New York, 1985.

[4] Bush, M.W., The Software Product Assurance Metrics Study: JPL's Software Systems Quality and Productivity, JPL Publication 89-6, February, 1989. Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California.

[5] JPL D-4013: Software Product Assurance Standard, Version 1.1 (Draft), August, 1988. Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California.

[6] Fagan, M.E., "Advances in Inspections," IEEE Transactions on Software Engineering, July, 1986, pp. 744 - 751.

APPENDIX A

ANNOTATED OUTLINE OF THE SPA PLAN

1. PURPOSE AND SCOPE OF THE SPA PLAN

This section describes the purpose and scope of the Software Product Assurance Plan developed for the specific project.

The purpose of each SPA plan is to establish and document the Software Product Assurance Program for the specific project. The plan should identify the scope of the SPA support to be provided by the independent SPA organization at JPL, Section 522.

2. REFERENCE DOCUMENTS

This section provides a complete list of documents referenced in the text of the plan.

2.1. Standards

This paragraph lists software standards which apply to the specific project. A typical list would include:

JPL:

D-4000 thru D-4014 series: JPL Software Management Standard Package

IEEE:

730-1984: Standard for Software Quality Assurance Plans

DOD:

(1) DOD-STD-2167A: Defense System Software Development
(2) DOD-STD-2168: Defense Systems Software Quality Program

NASA:

(1) D-GL-14: Software Quality Assurance for Project Managers
(2) DID-05: Software Product Assurance Plan
(3) DID-55: Software Quality Assurance Plan

OTHER:

Other standards accepted by the project.

2.2. References

This paragraph lists all other documents referenced in the SPA Plan.

3. CHANGE CONTROL OF THE SPA PLAN

This section describes the procedures to change the SPA Plan. The SPA Plan is a living document and is updated whenever changes are agreed upon. The changes made to the SPA Plan will be negotiated and approved by the Project Management, and SPA Manager. The change log form should be included after the title page of the specific SPA Plan.

4. PROJECT DESCRIPTION

The paragraphs of this section give a brief description of the project. The description addresses the project mission, major systems and subsystems, software components, and all software products covered by the SPA Plan. They also show organizational structure of the project with assigned responsibilities.

4.1. Technical Description

This paragraph describes the project mission, major systems, subsystems, and software components. All software components and program sets, with assigned degrees of criticality, and corresponding software products should be identified in detail so that Project Management can plan and schedule milestones, deliverables and reviews relating to Software Product Assurance.

4.2. Software Products Covered by the SPA Plan

This paragraph identifies these software components, program sets and corresponding software products which are covered by this plan.

4.3. Project Organization

This paragraph describes project organizational structure. It provides a brief description of each element of the organization together with delegated responsibilities. It also includes organizational charts showing interfaces to other organizations and reporting to the project structure.

4.3.1. Project Management of the Software Life Cycle

This subparagraph provides a description of the tasks associated with the portion of the software life cycle covered by this plan. It identifies specific organizational elements responsible for each of the tasks.

4.3.2. Independent SPA Organization

This subparagraph describes an independent SPA organization. The SPA Manager should be identified, and the reporting mechanisms to the project should be described.

4.4. Project Standards, Procedures and Conventions

This paragraph identifies software standards, and describes, or summarizes, procedures and policies required by, or applicable to, the project. It covers software documentation standards, programming languages to be used, logic structure standards, coding standards, commentary standards, and waiver procedures. If this information is provided in other documents, the references should be listed. The deviations from these standards, if any, should be described and justified.

4.5 Tools Techniques and Methodologies

This paragraph identifies the special software tools, techniques, and methodologies employed on the specific project that support Software Product Assurance tasks. It should state their individual purpose, and describe their use.

4.6 Assumptions, Limitations and Constraints

This paragraph describes the circumstances (assumptions, limitations, constraints) affecting the scope of the SPA support for the project.

In general, the scope of SPA support is influenced by many factors including the project's current stage in the development life cycle, specified software criticality classifications, budgetary and resource constraints, and development priorities. Each SPA Plan will be tailored to match the unique characteristics of the individual project.

5.0 SOFTWARE PRODUCT ASSURANCE ACTIVITIES

The paragraphs of this section describe all of the SPA activities to be performed for the project by the independent SPA organization.

5.1 Technical Evaluations

This paragraph describes technical evaluations of software products performed by the independent SPA organization. It includes a list of technical evaluations and corresponding software products (documentation and code) applicable to the project. This list should be prepared by the Project Management and SPA Manager. It should include all documents required for a specific software class as shown in D-4000, Table 3-4. All documents required for Class 1 software which should be evaluated by SPA are listed in D-4013, Section 3.2. SPA will support document preparation by providing "in-process" evaluations and recommendations. Document evaluation criteria will be addressed in JPL D-4001 thru D-4014 series and in the document inspection checklists.

5.2. Formal Inspections

This paragraph provides list of the formal inspection types applicable to the project and supported by the independent SPA organization. The level of the SPA support (e.g. providing a moderator, training, consultation) should be described. For mission critical software (Class 1), all seven types of inspections are recommended (see D-4013, Section 3.3.1).

SPA support in implementing inspections includes:

(1) Training project managers on inspection process.

(2) Training project development personnel as participants or moderators.

(3) Assisting in planning and scheduling inspections.

(4) Providing moderators for inspections.

(5) Documenting inspection results and ensuring that corrective actions are taken.

(6) Collecting and evaluating measurement data on the inspection process.

As a minimum, SPA should ensure that:

(1) Inspections are properly planned and scheduled.

(2) Adequate training is supplied to the moderators and participants of formal inspections.

(3) Defects are identified, recorded, and corrected.

5.3. Standards Compliance

This paragraph describes a mechanism used by the SPA organization to monitor and assure the compliance of software products and processes with project required standards and procedures.

SPA will evaluate compliance with established standards, practices and procedures required by the specific project. Normally, this is done during all SPA activities throughout the software life cycle.

5.4 Configuration Management

This paragraph defines the level of the SPA support in the planning, designing, implementing, and evaluating of the Software Configuration Management System. In addition to SPA activities required by D-4013, Section 3.4., SPA may assist in the selection of the Configuration Management tools and preparation of procedures, or evaluate the existing SCM tools and procedures to ensure that:

(1) The set of documents and code kept by SCM for each system represents a complete and current baseline.

(2) Traceability information between the current and previous versions of a baseline will be readily available and accurate.

(3) Only authorized changes will be implemented, and all affected systems and subsystems will be notified of proposed, pending, and accomplished changes.

5.5. Testing

This paragraph defines the level of SPA support in planning, designing, implementing, and evaluating software testing processes.

As a minimum SPA will:

(1) Evaluate test plans and procedures for compliance with appropriate JPL (D-4000) and project standards.

(2) Evaluate test requirements with respect to feasibility, adequacy, traceability to SRD and SIS-1, and satisfaction of software requirements.

(3) Evaluate effectiveness of software testing tools and testing environment.

(4) Evaluate adequacy and completeness of test cases.

(5) Evaluate traceability of test cases and test results to software requirements.

(6) Witness test activities to ensure that test procedures are followed.

(7) Assess and certify test results to ensure that all requirements are demonstrated as stated in the test plan.

(8) Ensure that all discrepancies and anomalies are documented and corrected.

(9) Witness and evaluate final validation activities to ensure the quality of the software product in terms of its operational requirements, readiness to perform its mission.

5.6. Metrics

This paragraph defines the level of the SPA support in the selection, evaluation, collection, maintenance and analysis of the project quality metric data. The list of specific metric data to be collected by SPA should be provided.

SPA will prepare or advise on and evaluate project's metrics plan. SPA will collect, archive, and analyze product and process metrics to determine the quality of the products being produced as well as the efficiency with which they are developed.

5.7. Tools, Techniques, and Methodologies

This paragraph identifies the SPA role in selection and evaluation of software tools, techniques, and methodologies employed on the specific project that support Software Product Assurance tasks.

5.8. Problem Reporting and Resolution

This paragraph describes the SPA role in problem reporting and resolution during the project software development. The project problem reporting system and procedures should be described.

SPA can support any of the JPL problem and resolution systems presently existing. These include Problem Failure Report (PFR) and Incident/Surprise/Anomaly (ISA) systems, the Failure Report/ Failure Accountability Report (FR/FAR) system in support of the Flight Project Support Office, and the Discrepancy Report (DR) in support of DSN.

SPA will ensure the following:

(1) Correct identification of problems and failures

(2) Technical adequacy of problem/failure analysis

(3) Validation of corrected discrepancies

(4) Proper closure of reports.

SPA will collect, archive, and analyze software problem and resolution data. By performing trend analysis of reported problems and their resolution, SPA can assist the project in identifying the problem areas and software components which need to be closely monitored.

5.9. Status Evaluations

This paragraph defines the SPA role in the Project status evaluation process through: project analysis, risk assessment, and participation in project (formal and informal) technical and managerial reviews. All project major reviews and corresponding schedules should be listed.

SPA will participate in all formal milestone reviews, and evaluate the product's readiness to proceed to the next development phase. The required milestone reviews are listed in D-4000. SPA will also participate in project status reviews and action meetings. Prior to any of the project major software reviews, SPA should receive a review package (at least a week in advance) to identify any software critical areas which should be discussed during the review. SPA will

integrate information from all its quality assurance activities on an ongoing basis, and provide Project Management with an overall project evaluation.

5.10. Contractor/Vendor Software Product Evaluations

This paragraph describes the provisions for assuring that the contractor/vendor developed/provided software products meet the established (contractual) requirements. As a minimum, the SPA activities in support of evaluation of contractual requirements, contractor supplied Software Quality Assurance Plan, and their implemcntations should be described.

In particular, SPA will support and contribute to:

* Requests for Proposals (RFP)
* RFP Review
* Proposal Evaluation
* Contractor Selection
* Fact Finding Negotiations
* Contract Review (Statement of Work, Contract Data Requirements Lists, Data Requirements Document)
* Evaluation of Software Quality Requirements
* Evaluation of Software Quality Assurance Plan
* Monitoring Contractor's Software Quality Assurance Activities
* Evaluations of Deliverable Software Products

The scope of SPA support depends on many factors including: classification of the developed software (degree of criticality), comprehensiveness of the contractor's Software Quality Assurance Plan, and results of previous SPA evaluations of the contractor/subcontractor software development programs.

5.11. Project Assessments

This paragraph described project assessments to be performed by SPA, as negotiated with the Project Management. SPA will lead an assessment team to analyze project's software engineering process to identify key areas for improvements, and to recommend actions that will lead to improvements.

6. SPA TASKS AND SCHEDULES

This section specifies current SPA tasks, all receivables and deliverables, and corresponding schedules. This section will be updated to reflect any technical, budget, and resource changes on the specific project. This information is usually included in the SRM package and may be used in this section.

7. RISKS AND CONTINGENCIES

This section describes the risks and contingencies associated with the technical, budgetary, and resource changes on the specific project.

8. APPENDICES

APPENDIX 1. ACRONYMS

This appendix lists all the acronyms used in the specific SPA Plan.

APPENDIX 2. GLOSSARY

This appendix contains a glossary of terms used in the specific SPA Plan.

Appendix B

Recommended Product Assurance Activities

SOFTWARE PRODUCT ASSURANCE ACTIVITIES

The SPA organization evaluates all indicated documents and participates in all indicated reviews. Level indicates software criticality. A= Critical; E= Low Risk.

JPL SYSTEM LIFE CYCLE PHASES REQUIRED BY JPL D-4000	A - LEVEL	B - LEVEL	C - LEVEL	D - LEVEL	E - LEVEL
o SYSTEM REQUIREMENTS ANALYSIS*					
- System Functional Requirements Document	Required	Required	Required	Combined with SRD	Safety-critical items only
- Formal Inspections	Recommended	Recommended	Recommended		
- System Functional Requirements	Required	Required	Required	Required	Required
- Software Management Plan (SMP)	Required	Required	Required	Required	Safety-critical items
- SMP Review	Required	Required	Required	Required	
- Work Implementation Plan (WIP)**	Required	Required	Included in SMP	Included in SMP	Included in SMP
- Risk Management Plan	Required	Recommended	Recommended		
o SYSTEM FUNCTION DESIGN*					
- System Functional Design Document	Required	Required	Required	Combined with SRD	Safety-critical items only
- Formal Inspections	Recommended	Recommended	Recommended		
- System Functional Design Review	Required	Required	Required	Required	Required
- System Integration and Test Plan (VOL. 1)	Required	Required	Required	Combined with SRD	Safety-critical items only
- Formal Inspections	Recommended	Recommended	Recommended		
- Software Configuration Management Plan (SCMP)	Required	Required	Included in SMP	Included in SMP	
- Subsystem Work Implementation Plan Review	Required	Required	Required	Required	
- Software Product Assurance Plan	Required	Required	Included in SMP	Included in SMP	

*Required for software intensive systems.
**Work Implementation Plan is required for each phase of the life cycle.

SRD - Software Requirements Document
SMP - Software Management Plan

Software Product Assurance Activities

	A - LEVEL	B - LEVEL	C - LEVEL	D - LEVEL	E - LEVEL
o SUBSYSTEM REQUIREMENTS ANALYSIS*					
- Subsystem Functional Requirements	Required	Required	Required	Included in SRD	Safety-critical items only
- Formal Inspections	Recommended	Recommended	Recommended		
- Subsystem Functional Requirements Review	Required	Required	Required	Required	Required
o SUBSYSTEM FUNCTIONAL DESIGN*					
- Subsystem Functional Design Document	Required	Required	Required	Included in SSD	Safety-critical items only
- Integrated Software Functional Diagram	Required	Included in SSD	Included in SSD	Included in SSD	Safety-critical items only
- Formal Inspections	Recommended	Recommended	Recommended	Recommended	
- Subsystem Integration & Test Plan (Vol. 1)	Required	Required	Required	Included in SRD	Safety-critical items only
- Formal Inspections	Recommended	Recommended	Recommended		
- Subsystem Functional Design Review	Required	Required	Required	Required	Required
- S/W Work Implementation Plan Review	Required	Required	Required	Required	
o SOFTWARE REQUIREMENTS ANALYSIS					
- Software Requirements Document (SRD)	Required	Required	Required	Required	Safety-critical items only
- Software Interface Specification (Vol. 1)	Required	Required	Required	Included in SSD	Safety-critical items only
- Formal Inspection (SRD & SSD-1)	Required	Recommended	Recommended	Recommended	
- Software Requirements Review	Required	Required	Required	Required	Required
o SOFTWARE DESIGN					
- Software Specification Document (SSD Vol. 1)	Required	Required	Required	Required	Safety-critical items only

*Required for Software intensive systems. SRD - Software Requirements Document SSD - Software Specification Document.

SOFTWARE PRODUCT ASSURANCE ACTIVITIES					
	A - LEVEL	B - LEVEL	C - LEVEL	D - LEVEL	E - LEVEL
- Software Interface Specification (Vol. 2)	Required	Required	Required	Included in SSD	Safety-critical items only
- Formal Inspection	Required	Recommended	Recommended		
- Software Test Plan (Vol. 1)	Required	Required	Required	Included in SRD	Safety-critical items only
- Formal Inspections	Recommended	Recommended	Recommended		
- User's Guide/Software Operator's Manual (UG/SOM)	Required	Required	Recommended		
- Software Design review	Required	Required	Required	Required	Required
o SOFTWARE IMPLEMENTATION AND ACCEPTANCE TEST					
- Subsystem Integration & Test Plan (Vol. 2)	Required	Required	Required	Included in SRD	Safety-critical items only
- Formal Inspections	Recommended	Recommended	Recommended		
- Software Interface Specification (Vol. 2 Upgraded)	Required	Required	Required	Included in SSD	Safety-critical items only
- User's Guide/S/W Operator's Manual	Required	Required	Required	Final version if implemented	Safety-critical items only
- Software Specification Document (Vol. 2)	Required	Required	Required	Required	Safety-critical items only
- Formal Inspection	Required	Recommended	Recommended		
- Source Code Formal Inspection	Required	Recommended	Recommended	Safety-critical items only	Safety-critical items only
- Software Test Plan (Vol. 2 & 3)	Required	Required	Required	Included in SRD	Safety-critical items only
- Release Description Document (RDD)	Required	Required	Required	Included in UG/SOM only	Safety-critical items
- Software Transfer Agreement	Required	Required	Included in RDD	Included in UG/SOM	Safety-critical items only
- Fault Protection Algorithm Analysis	Required	Required	Recommended	Recommended	Safety-critical items only

SRD - Software Requirements
RDD - Release Description Document
UG/SOM - User's Guide/Software Operator's Manual

SOFTWARE PRODUCT ASSURANCE ACTIVITIES

	A - LEVEL	B - LEVEL	C - LEVEL	D - LEVEL	E - LEVEL
- S/W Sneak Analysis	As required for safety of personnel and/or equipment	As required for safety of personnel and/or equipment	As required for safety of personnel and/or equipment	As required for safety of personnel and/or equipment	As required for safety of personnel and/or equipment
- Implementation Status Review	Required	Required	Required	Required	
- Software Acceptance Review	Required	Required	Required	Required	Required
o SUBSYSTEM INTEGRATION, TEST & DELIVERY*					
- Subsystem Integration and Test Plan (Vol. 3)	Required	Required	Required	Included in SRD	Safety-critical items only
- System Integration and Test Plan (Vol. 2)	Required	Required	Included in RDD	Combined with SRD	Safety-critical items only
- Formal Inspections	Recommended	Recommended	Recommended		
- Subsystem Transfer Agreement	Required	Required	Included in RDD	Included in UG/SOM	Safety-critical items only
- Subsystem Delivery Review	Required	Required	Required	Required	Required
o SYSTEM INTEGRATION, TEST & DELIVERY*					
- System Integration & Test Plan (Vol. 3)	Required	Required	Required	Combined with SRD	Safety-critical items only
- System Transfer Agreement	Required	Required	Included in RDD	Included in UG/SOM	Safety-critical items only
- System Delivery Review	Required	Required	Required	Required	Safety-critical items only
o SPA ACTIVITIES THROUGHOUT SOFTWARE LIFE CYCLE					
- S/W Criticality Evaluation	Required	Required	Recommended	Recommended	Safety-critical items only
- Configuration Management Evaluation	Required	Recommended	Recommended	Recommended	Safety-critical items only
- Metrics (Plan, Collection and Analysis)	Recommended	Recommended	Recommended	Recommended	Safety-critical items only
- Testing Evaluation	Required	Recommended	Recommended	Recommended	Safety-critical items only

*Required for software intensive systems.

SRD - Software Requirements
RDD - Release Description Document
UG/SOM - User's Guide/Software Operator's Manual

SOFTWARE PRODUCT ASSURANCE ACTIVITIES					
	A - LEVEL	B - LEVEL	C - LEVEL	D - LEVEL	E - LEVEL
o MISSION OPERATIONS					
- Problem Failure Reporting System (PFRS) Management & Assessment	Required	Required	Required	Required	Required
- Risk Assessment	Required	Required	Required	Required	Required
- Operations Training Assessment	Required	Required	Required		
- Problem/Failure Trend Analysis	Required	Required	Required		
- Downlink Assurance	Required	Recommended	Recommended		
- Configuration Management Assurance	Required	Required	Required		
o COMMAND ASSURANCE					
- Command Assurance	Required	Required	Required	Required	Required
- Command Awareness Training	Required	Recommended	Recommended		

Effects on SQA of Adopting C++ for Product and Tool Development at Apple

Peter M. Martin*, Eric W. Anderson, Richard K. Dizmang
Apple Computer, Inc.

With Apple having committed to support C++ on the Apple Macintosh computer, software engineers elected to revise the Macintosh shell application, *Finder*, to be rewritten in C++ in an object-oriented manner. SQA engineers elected, in turn, to develop a more rigorous testing process of the revised *Finder*, utilizing the object-oriented facilities of C++. The existing feature-based approach to testing was augmented by an early analysis of source code, in order to add code for testing purposes. This code, primarily in the form of messages sent to a specific testing class hierarchy, allowed the low-level testing of features to substitute for user level testing, and to facilitate a conversion to more automated testing of a highly user-interactive application. By emphasizing the object-oriented use of C++, the development engineers produced a higher quality product that was much easier to test, and the test engineers developed a more effective and reliable testing process.

Peter M. Martin is a Team Leader for Macintosh System Software Quality and Testing. He has designed and implemented a testing program for measuring perfomance in medical imaging, as a physicist with a CT scanner manufacturer, and has conducted academic research in nuclear medicine and biophysics. He has developed and applied computer programs and diagnostic procedures for CT scanner data acquisition, and analysis of electron beams and CT images.
M.S., Nuclear Engineering, U.C. Berkeley; B.S., Physics, Stanford

Eric Anderson is a Test Engineer for Macintosh System Software Quality and Testing. He has contributed to a commercial dynamic CAD graphics application, and is currently developing tools to test operating system services in the next release of Macintosh system software.
B.S., Comp.Sci. and B.A., Physics, Sonoma State University

Richard Dizmang is a Test Engineer for Macintosh System Software Quality and Testing. He has been an assembly-level software developer on the entire Apple product line since 1977, including two commercial products for the Apple II; he is currently a lead developer for Macintosh OS test tools.

* To whom correspondence may be addressed; all authors may be contacted at:

Apple Computer, Inc.
10500 N. De Anza Blvd. M/S 27-BA
Cupertino, CA 95014

Effects on SQA of Adopting C++ for Product and Tool Development at Apple

Peter M. Martin, Eric W. Anderson, Richard K. Dizmang
Apple Computer, Inc.

Introduction

The addition of C++ as an Apple-supported programming language and its use in developing production code has posed new challenges and opportunities to the Software Quality Assurance (SQA) organization at Apple Computer. The technical qualifications of SQA engineers who are involved in testing have become much greater: they must be more versatile programmers, so that the object-oriented potential of C++ can be utilized for more objective and more easily automated testing. Managers in both SQA and in Software Engineering have also responded to two additional challenges. One of these has been to accept the costs, such as increased conformance to programming standards, in order to reap the benefits, such as more maintainable and testable code, whenever software engineering projects rely on C++. The other challenge has been to recognize that using C++ to best advantage means that proper design becomes more important than it used to be, and that easier testing would result from involving QA engineers to a greater extent during the design phase.

Apple's development system for the Macintosh®, *Macintosh Programmer's Workshop* (MPW), currently supports C, Pascal, and 68000 assembly language. After it became known that Apple would be supporting C++, software engineers began a major revision of the system application, *Finder*™, into C++ for the next release of Macintosh system software, System 7.0. Shortly thereafter, SQA engineers decided to approach the testing of *Finder* differently, developing new C++-based tools, and participating more in its design and implementation phases. They also decided to use the experience gained in this instance as an opportunity to establish new testing procedures for other C++-based software development efforts at Apple. SQA management was informed of the engineers' decisions, and supported them.

Background

History

The primary responsibility of SQA at Apple Computer is to test all of the software provided by Apple for the Macintosh and Apple II computers, as well as for Apple peripheral products. Extensive testing is also conducted with third-party software, in order to assess and document the compatibility of selected applications whenever the functionality of system code changes or becomes more stringent. Although the testing organization is an Apple operation, its makeup and management are actually separate from the software engineering group that design and write production code. The comparison to an in-house contract service for testing is sometimes made.

The methodology for software testing at Apple was originally operational, and was applied only to code delivered with whatever documentation the development engineers made available. The code was treated as a "black box" of services, invoked by third-party applications or by test tools which were often developed by the engineers themselves. Testing was designed and performed largely by experienced users. Even testing the application programmer's interface (API) was presumed to be sufficient by exercising at the user level selected applications that used the features of the API. Faults were reported as "bugs" by means of an in-house document control system, and their repair was tracked throughout the development process.

Present Procedure

Against the backdrop of a seemingly adequate but variable testing process a decision was made by SQA management to approach testing in a more formal and structured way. The current procedure, which is independent of the language for software development, calls for a written specification to be provided by the development engineers for each project that is part of a "product", as the customer might perceive it. The procedure starts with the engineering specification, which explains what services are to be provided to the developer or user, how the services are invoked, and what error conditions are reported. Based on the specification, the test engineer prepares a list of the features, a list of the tests to

be applied to each feature, and a test plan that describes how the testing project is to be conducted. The test plan may also indicate that a test tool may need to be developed as an engineering project internal to SQA. This specification-based process is intended to alert Engineering and SQA managers to address other potential issues during the development cycle in sufficient time for testing.

The structured testing approach continues to be the central methodology by which the policy to assure software quality at an early stage in the testing process is implemented at Apple. It does not set any standards or rely on any programming rules. It is simply a formalism for project management during product development, and is based on the features of the anticipated software, as declared by the software engineers in the design phase. It does not call for review of product design or implementation strategies, design reviews or code reviews, or preliminary testing by the development engineers.

Formal testing starts with the receipt of testable object code by SQA. Preparation for testing, however, occurs well before object code appears. The SQA engineers who develop test tools for new software products may become involved in design reviews, as time allows and as their relationship with the development engineers solidifies. They have regular testing and administrative responsibilities, as well.

Two factors of the software environment on the Macintosh facilitate the adoption of the object-oriented programming (OOP) paradigm, and are often used as metaphors. The extensive Toolbox, contained partly in ROM and partly in RAM, is described as a set of managers, with a large number of procedure calls -- a large run-time library. All Macintosh developers are presented with these chunks of functionality that should be used in all applications, in order to provide a consistent human interface and, by the way, encourage the re-use of code for new applications. This predisposes them to think in terms of objects with complex, but well-defined services. In addition, a goal of Apple is to provide with its products an intensely personal experience, which results in Macintosh applications being highly interactive and event-driven. Applications should use a top-level structure, therefore, in which windows and pull-down menus are managed, while internal dispatching is used heavily for program control. This experiential model contributes more to the notion of sending messages to objects than to passing parameters to procedures.

Finder for System 7.0 provided the first practical opportunity for SQA to implement a testing methodology appropriate for an application written in an object-oriented way. Although C++ provides the major properties of an object-oriented programming language, it is in fact a superset of C, and as such allows the programmer to regress to procedural programming. The project was undertaken with the understanding that the software engineers would exploit an object-oriented paradigm in their use of C++.

Goals and Objectives

In revising and enhancing the approach to testing *Finder*, the major goals were to render the product more "testable", to determine observable measures of testability, and to describe the material gain to the company that was realized. A conscious effort was made to improve just testability, among the many quality attributes that can be attached to software in general. While the impact on other measures of software quality was not ignored, the overriding focus was clearly on testability, or "verifiability", in the parlance of Deutsch [Deutsch, 1988].

Another goal was to establish a more reliable and reproducible testing process that could be applied to other software projects. Typically, much of the testing of the Macintosh system and applications relies on screen actions and user judgments, which allows more variability than desired. Because *Finder* has a substantial human interface component, which has been used extensively as a test platform on the Macintosh, it served as a good model. Whereas in the past, the application of test criteria often required a user to make observations and report that proper screen behavior occurred, the new process was intended to use code incorporated in the product for testing purposes, in order to reduce reliance on human judgment. The prospect of automated testing also became more likely if test code for the purpose could be included. It was generally believed that test code could be combined more safely with production code in an OOP language than code written in a procedural language. Including test code with production code during development was expected to:

1) allow manual and automated testing;
2) foster test case generation by using a recording/playback mechanism with editing capabilities;
3) invoke built-in tests through normal usage;

4) reduce the need for human interaction and subjective verification;
5) standardize the testing process, independent of hardware configuration.

This effort was also an attempt to take the structured testing approach as far as possible within the object-oriented framework. The move from a heuristic approach to testing, which was used in the early days of the Macintosh, to a more structured approach can be characterized as follows:

Testing process:	**STRUCTURED**	**HEURISTIC**
Product definition:	**Specified**	**Operational**
Approach:	**Deterministic**	**Exploratory**
Tester bias:	**Analytical**	**Curious**
Study question:	**"What does it do?"**	**"What if I try this?"**

Table 1.

A corresponding characterization to admit the OOP paradigm is described by analogy, as an adequate vocabulary does not seem to be in common use yet. A distinction between controlled laboratory experiments and partially controlled field observations is drawn:

Sciences analogy:	**LAB WORK**	**FIELD WORK**
Central activity:	**Manipulation**	**Observation**
Process:	**Quantitative**	**Qualititative**
Approach:	**Prescriptive**	**Descriptive**
Attitude:	**Believe provability**	**Accept ambiguity**
Study question:	**"How does it work?"**	**"What do I see here?"**

Table 2.

A strict comparison of object-based testing, relative to structured or heuristic testing, or to the distinction between lab work and field work is not intended. Rather, the analogy is meant to show the increased pursuit of rigor and definition in the testing process, as an increased degree of manipulation and prescription are imposed, with the associated hope that the quality of the targeted software is indicated with greater definition. The

new process is not intended to replace final testing of the product to be shipped, but rather to insure product robustness and to accelerate the revision process during development.

Another goal of this effort was to be evolutionary within the organization by creating a practical example of making the transition from a mindset of quality assurance to one of quality engineering. The critical element in moving from "assurance" to "engineering" was seen to be in assuming a role in influencing product design according to some general principles for quality. The critical qualification of members of the QA organization was to demonstrate software engineering skills that would generate an appreciation of their role in design reviews.

Strategies

Maps and source code analysis

Good object-oriented programming style was expected to make the analysis of source code much easier, and thereby make the design and implementation of a testing strategy easier. A fundamental prerequisite for developing the testing strategies, therefore, was to have access to source code. SQA engineers at Apple do not review source code as a rule, so such access is usually not an issue. But the prospect of source code analysis for structural features, use of globals, coding style, dependencies, and flow control was expected to be sufficient justification.

With C++, it is possible to provide some of the source code to facilitate testing, yet without revealing all of the implementation details. (The value of doing so is considered later.) This can be done by providing only the class and method definition files ("header files"). A corresponding requirement in doing so is that engineers do not add new definitions in the implementation files if those definitions are not accessible to engineers in SQA.

The creation of several kinds of "maps" was addressed by studying the source code. Since the maps could be determined mechanically, special tools were designed to produce them automatically. A tree structure of the class hierarchy, a cyclic graph of method invocation from within other classes, and a message flow diagram, ignoring run-time branching, were

each to be provided by source code analysis tools. The extent of overriding methods and operators ("polymorphism") was also tabulated.

Programming standards for testability

C++ allows rules to be broken as much as it offers a formalism for clarity and modularity in programming. Its implementation at Apple, intended to be ANSI-compatible, also has some Macintosh-specific enhancements. That some constraints in its use were needed was generally believed. A collection over the course of the development effort of a testability-focused "elements of style" was planned, as much to refer to features of the C++ language to be exploited or avoided, as to encourage or discourage such practices as overriding of methods, and referencing class variables directly.

Classes for run-time testing

To facilitate the automation of user level testing, special testing classes were designed and implemented through hooks added to the source code. These testing-specific classes were designed to provide services dedicated to testing objectives, such as providing an assertion mechanism, verifying successful completion of prescribed actions and journaling the outcome.

In order to utilize a hierarchy of classes that were to be designed for testing purposes, mechanisms for sending messages to them were needed. One obvious technique was to override the C++ operators, "new" and "delete", for every instance of object construction and destruction. One objective, therefore, was to find other practical mechanisms to send messages to objects included for testing.

Tool development for automation

The most far-reaching effect of this effort was anticipated to be the establishment of a viable technique for automated testing of an application with an extensive user interface. The interactive nature of using a Macintosh is a blessing to the user, but a bane to the tester. By providing ways to signal that the screen has been updated in a certain way, and that the desired action has been verified, a new technique could be exploited for yet other

tools to drive the testing process, without user intervention. The risk of excessively perturbing the very code to be tested was something to be assessed through experimentation.

Pursuit of this strategy represented the most significant departure from the structured testing process that was being applied elsewhere. Typically, the definition of the product to be tested came from the features identified in the engineering specification. These features could be services provided to the user through mouse clicks or keystrokes, or to the application developer, through an API. However, automation in this case required that certain points in the source code be identified as places where the action associated with the "feature" in question was effectively complete, so that the appropriate messages could be sent for testing purposes.

Other measures of software quality

Since improved testability was the major goal, *Finder* was treated not as a set of code modules, but as a set of features, which were assumed to be independent at the most elemental level. This operational view of the product kept us from considering a number of other measures of quality. The concepts of structural analysis and data flow diagrams [DeMarco, 1978] were not applied directly to the source code for *Finder*, but were transformed into an analysis based on hierarchy and invocation maps, for the sake of identifying test code insertion points. Similarly, code thread analysis, *i.e.*, checking for frequency of code usage, was not considered. Neither did we consider metrics based on the number of lines of code [Halstead, 1977] that might be useful for an object-oriented implementation. Although the association of objects with features is promoted in an application written in an OOP language, improvements in testability were not assumed.

The "level of granularity" with which the source code was studied in this effort also restricted the scope of possible metrics for quality. Improving source code was not the objective; however, as stated above, assuring proper functionality was an objective. The intention was to use production source code as a tool itself in the testing process, just as it was also the source of the functionality to be tested.

Results

Training

SQA engineers had to develop a proficiency in C++, as demonstrated by the creation of a test tool written in C++, in order to assure the development engineers of sufficient competence, and to carry on technical dialogue throughout the development process. Aided by an in-house training course for C programmers wanting to learn C++, the authors attained respectable proficiency in approximately 2 months.

Maps and source code analysis

Source code analysis was implemented in a two-stage process. Parsing and cataloging tools were used to generate intermediate text files containing lists in a database-like format, suitable for import into an Excel® spreadsheet or a HyperCard® stack. Then presentation tools, such as MORE II®, transformed the text files into a tree structure layout, or other representation. This approach provided two facilities: an expandable means to develop an analysis tool without being coupled to the development of a presentation tool, along with a way to retain the simplicity of flat-file database concepts in managing the intermediate text files. The parsing and cataloging tools were first attempted within the programmer's development environment, MPW Shell. The contents of some of the intermediate text files are shown in Table 3.

File*	Item 1	Item 2	Nature of information
D	Class	Parent	classes and inheritance; tree structure
D	Class	Method	catalog of available services
D	Method	Method	method-method dependencies
D	Class	"operator"	operator overrides
D	Class	Variable	data structures
I	Class	Other class	instantiation tree
I	Class	Methods' class	class referral graph
I	Class	Method	method flow diagram; invocation tree

* D = Definition; I = Implementation

Table 3.

For either the definition file or the implementation file, the first column contains a class name, while the second column contains the item of interest that is referenced in the named class.

A tool to generate the Class/Parent text file was completed in one day. Figure 1 shows a partial class hierarchy produced by applying it to *Finder* :

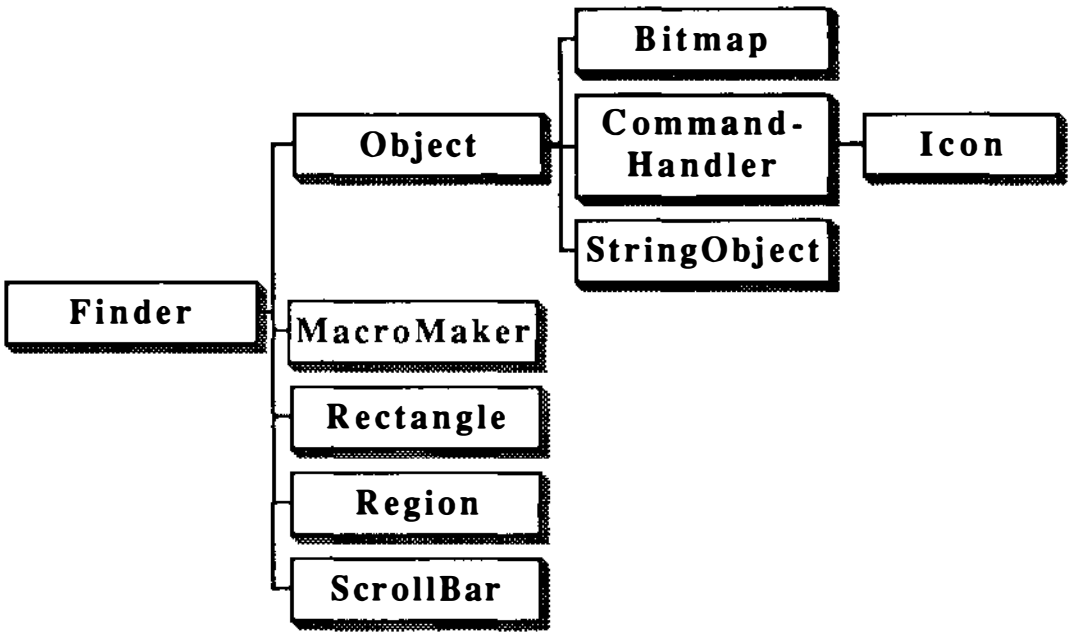


Figure 1.

One of the early findings from analyzing the class hierarchy was the observation that the class **`Finder`** was derived from an Apple-specific base class, **`HandleObject`**. The purposes of this class are to allow the use of the Macintosh Memory Manager from within an application and to allow the kernel to relocate memory objects for run-time efficiency. However, due to the implementation of the class, the consequence of the derivation was to deny the exploitation of multiple inheritance, for testing purposes or otherwise.

Programming standards for testability

SQA asked for special collaboration with the engineers during development. Some of the constraints were requested in order to facilitate source code analysis and to promote efficient testing by means of added code.

Additional constraints were requested for the sake of promoting class reusability and conceptual clarity as a matter of good object-oriented programming. The considerations asked for by SQA included: minimizing the depth of the class hierarchy, minimizing the extent of overriding operators and methods, grouping and minimizing the C code fragments that used a procedural programming style. The engineers concurred with the value of these requests.

Classes for run-time testing

A set of classes was developed as a standard package of services to be compiled with *Finder* for testing purposes. A partial tree structure for these classes is shown in Figure 2:

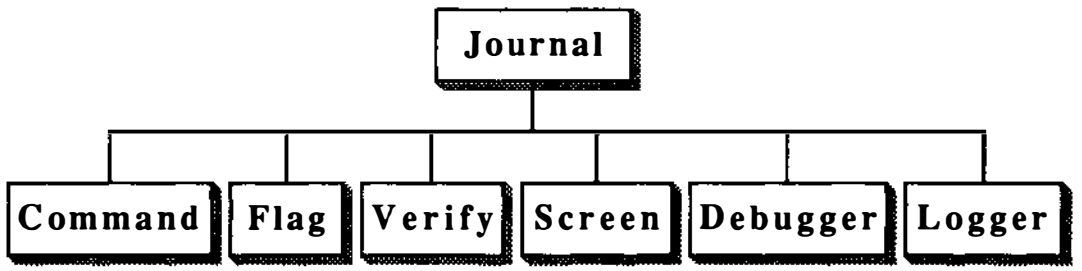


Figure 2.

The **Journal** class provides test-global services. The **Command** class provides a control mechanism once we are in our testing code, to include remote control for automation. The **Flag** class manages a set of state variables during a testing session. The **Verify** class receives messages from code injected into *Finder* at key locations -- the "hooks", to check for proper execution. The **Screen** class maintains a window and menu for user I/O with the test code during testing. The **Debugger** class maintains an inter-process communication (IPC) mechanism with the standard Macintosh debugging tool, *Macsbug*®. *Macsbug* can be invoked by the user or programmatically, and has a facility to "peek" at memory structures during program execution. It can also execute user-added code fragments. The **Logger** generates and stores information about **Journal** diagnostics and *Finder* actions at run time. The **Logger** output can also be used to recreate a testing scenario for full automation.

The major value of source code access was realized early in the implementation phase. The specification of an appropriate class hierarchy for test purposes led to the engineers themselves implementing and incorporating into the product the test classes that were proposed by the SQA engineers. This provision now allows code that was intended for use during product development to be left in the product for later use in the event that a possible fault needs to be investigated after shipment. Incorporation of classes for testing purposes was considered a more significant consequence than tool-assisted structural analysis for the code review process.

Automation

Because the test classes had provisions to allow tests to be driven by any generator of the appropriate events, the time needed to test *Finder* was reduced by more than ten-fold, in spite of an increasing number of CPU platforms on which testing was required. The `Command` and `Logger` classes were used to provide an internally consistent mechanism for automated testing, as well as to provide control of a remote machine from a master machine, and to apply test criteria to indicators of *Finder* actions.

Conclusion

The SQA engineers went beyond the testing procedure that was expected for a structured testing approach. They did so, however, for the sake of providing more testing, more automated testing and quicker turnaround times per development cycle than with typical user testing. Collaboration between the development engineers and the QA engineers was consistent throughout the project. This collaborative relationship among the development and the test engineers contributed to reaching appropriate agreements promptly, particularly with respect to the consequences of coding strategies. The protective posture toward access to source code was successfully accomodated, as well.

Corporate culture at Apple is a significant issue for its employees. That this project originated with SQA engineers and was supported by their managers and development engineers was recognized as an example of

Apple culture at work. Substantial latitude was provided in allowing front-line engineers to pursue new and promising processes. Several technical advancements were realized, such as the development of source code analysis tools, a testing class hierarchy, implementation of an automation strategy, and compiling modified production code for test purposes. While development engineers allowed more controlled access to source code, they received in return a more convivial process and more specific testing that gave them a higher confidence level in creating a reliable software product.

References

DeMarco, Tom. *Structured Analysis and System Specification,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

Deutsch, Michael S. and Willis, Ronald R. *Software Quality Engineering,* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

Halstead, M. M. *Elements of Software Science,* Elsevier North-Holland, New York, 1977.

Assuring the Quality of Contracted Software

George Stevens
ADVANCE Computer Engineering, Inc.
P.O. Box 10736
Portland, Oregon 97210
503-297-9261

Abstract

Based on experience with many contracted software projects, several key software quality themes are identified. Guidelines are provided to enhance the quality of contracted software.

Biographical Sketch

George Stevens is the founder and CEO of ADVANCE Computer Engineering, Inc., a Portland firm that provides contract software development and quality assurance services to high technology industry. During 1981 through 1986 he worked as a software engineer in several Beaverton, Oregon based high tech companies. George has an MS, Computer Science 1981; BS, Environmental Science 1974; BA, Sociology 1974 from Washington State University.

Assuring the Quality of Contracted Software

George Stevens
ADVANCE Computer Engineering, Inc.
P.O. Box 10736
Portland, Oregon 97210
503-297-9261

INTRODUCTION

More than ever before, today's high tech companies find their software being developed by "outside" sources. For example:

- Developers of commercial software packages.

  Commercial software packages get customized for integration into systems, or larger software products.

- Vendors of commercial software packages.

  Existing software gets bundled or integrated into systems products and requires custom software "glue" to hold the bundles together.

- Contract development and systems integration companies, and independent contractors.

  They supply services to customize existing software, develop custom software products, and integrate software and hardware into systems.

Fierce competition in the computer, software, and electronics industries encourages development of software on a contracted basis by "outside" sources. Sometimes there are significant competitive advantages to be gained by utilizing "outside" developers.

Examples are:

- Reduced time-to-market achieved by:

  1) Leveraging new software from existing software bases.

  2) Developing subsystems in parallel.

- Strategic alliances, partnerships and consortia.

- Leveraging off of other's technological expertise, rather than paying the time-to-market cost involved to develop, grow, or hire the required expertise in house.

As a result, today's software developers and managers need be concerned not only about the quality of the software and products we develop in our own organizations. We must be equally concerned about the quality of the software developed by "outside" sources. I hope to provide you with some clear guidelines in this paper to facilitate your address of this concern.

The guidelines I present are based upon my experience over the past 3 years with contract software development and quality assurance projects: Especially quality assurance projects aimed at acceptance testing of contracted software. During these projects, I've seen a lot of things go right and go wrong. I'd like to share my observations with you.

First, I have three overall observations about quality in contracted software projects:

- The quality of the software is often lower than we would wish for.
- Low quality often results in expensive, unpleasant situations that can slip schedules, lose market share, and even kill products or companies.
- Quality in contracted software projects CAN be assured, as evidenced by some very, very successful projects.

REPEATING LOW QUALITY THEMES

A number of themes keep popping up in contracted projects that result in low quality software. I generally define software quality as being measured by the:

- Number and severity of bugs.
- Adequacy of delivered software to perform the intended function.

Let's first take a look at these Low Quality Themes. Then I'll draw several conclusions, and summarize the themes in some guidelines that will help you assure the quality of any contracted software projects you become involved in.

Theme -- **What are we supposed to be doing, anyway?**

Low quality of software in performing it's intended function happens when no one is clear about what the software is supposed to do.

Good specification of deliverables usually result in quality software. So do mechanisms of controlling changes that inevitably arise during a project.

Theme -- **They were great technologists, but....**

Typically, we base many of our "hiring" decisions concerning contractors on their technological prowess. Of course this is important, but quality often suffers when estimates are unrealistic, schedules slip, the technologists dislike testing, or the contractor's business gets in financial or legal trouble.

Appropriate experience with the technology, plus good project and business management experience is necessary for quality results from contract projects.

Theme -- **I thought YOU were responsible for testing!**

This is the most popular theme of all. It's so easy to forget about testing. To assume the "other guy" will take care of it. If no one is responsible, it's not likely to get done. At least within schedule and budget constraints.

At the beginning of the project specify what kind of testing is to be done, who does it, and when it needs to get done.

Theme -- **Oops, the budget doesn't include testing!**

This is a popular variation on the above theme. It happens when people realize that testing is necessary and desirable, but are reluctant to pay for it. Often, this occurs at the end of the project. Just in time to blow the project budget, schedule, or result in more bugs than desirable.

Make sure adequate testing is included in the project budget at the start of the project, regardless of who is responsible for doing the testing.

Theme -- **It's done now, isn't it?**

> "Uh, no. Change the data display and make all the error codes less than zero".

It's done now, isn't it?

> "Almost, but will you make it run on a Sun too"?

Specifications and acceptance criteria specify what "done" is AND the acceptable level of quality in the final deliverables. A way of controlling changes to

these is also worthwhile.

Theme -- **But, they said they had the proper experience and commitment to quality.**

Low quality results when a contractor has a reputation for low quality, regardless of their technical experience. Talk is cheap.

Check the contractors reputation and track record through references from their past projects.

Theme -- **What the heck is going on now?**

A contracted software project is a joint effort. It involves at least two parties that have to cooperate to get things done. The second most frequent theme is miscommunication between the 2 parties.

Regular status reports and meetings result in better quality since problems can be detected before they become critical. Team work and good communication are essential to produce high quality contracted software.

Theme -- **It says right here in the contract, paragraph 42.A, that....**

By the time people starting pointing at clauses in the contract, things have gotten out of control. Try to set up a Win/Win situation from the start.

Things that start out wrong, seldom end right. All of the low quality themes focus upon issues that were not adequately addressed when a contract software project was started.

The key to quality in contracted software is to structure the project at the beginning to produce quality results.

GUIDELINES FOR ASSURING QUALITY IN CONTRACTED SOFTWARE

The following guidelines form a check list you can use to structure a contract software project to produce quality results.

At The Start of The Project

Develop Workable Specifications and Plans

Requirements and detailed specifications exist, or will be developed as a first part of the project.

A mutually agreed upon way of controlling the changes in specifications that inevitably arise during implementation is setup at the beginning of a project.

Prototyping critical areas is planned for.

A complete project plan and budget are made, then tracked and maintained.

Integration time and testing is included in project plan and budget.

Deliverables resulting from the project are clearly specified, including quality targets and acceptance criteria.

The resources needed, and which party supplies them, is part of the project plan.

The people needed, the party to provide them, their responsibilities and working relationship are part of the project plan.

Contractor's Capabilities and Reputation

Appropriate experience with the technology.

Track record of successful project management.

Track record of successful business management.

Good cultural fit between contractor and your organization.

Contractor has a reputation of delivering high quality results and standing behind their work.

Contractor wants to meet your objectives and priorities, rather than "hidden agendas".

Mutually beneficial business interests exist beyond the immediate project at hand.

Testing the Contracted Software

Who's responsible for the testing? This should be clearly spelled out.

What kind of testing is necessary? What is the scope, breadth, and depth required? There should be a test plan addressing these issues.

Is test suite development necessary?

What is it? Who does it? Who runs it? Who maintains it? Who pays for it? Who owns it?

When is testing to be done? At what phases of the project?

What's a bug? What's a feature? How are results of tests used and who sees them?

How are problems resolved and tracked? Who fixes a bug, when, and then what?

Project budget and plan should include all testing as a separate item, task or set of tasks.

Acceptance criteria and acceptance testing is included in the project and used to determine what "done" is. Acceptance should include an acceptance period.

During The Project

Budget, schedule, and progress reports are provided periodically.

Contractor/Contractee meetings occur periodically.

Key people in Contractor and Contractee organizations are identified early on, with clearly specified responsibilities and relationships.

The agreed upon Change control process is used.

At the End of The Project

Required level of documentation has been planned, specified, budgeted for, and delivered.

Format and media of deliverable software has been planned for and delivered.

Someone has been identified to take care of maintenance of the contracted software during and after the acceptance period.

Archival: Someone has been identified to pull together all software, documentation, notes, and other items necessary to facilitate cost effective maintenance and enhancement once the project is complete.

Like any other complex endeavor that yields high quality results, contracted software can be high quality when:

- The appropriate organization of the effort is done with thought and care.

- Sufficient people and resources, of the right kind, are committed to the project.

I hope these guidelines will be of use to you on your current contracted project, and the ones just over the horizon.

VERIFICATION OF SOME PARALLEL ALGORITHMS

Dennis de Champeaux
HP-Labs
1501 Page Mill Rd, 3U
Palo Alto, CA 94304-1181

1989 June

Abstract

We consider parallel algorithms that are formulated as distinct cooperating processes which do *not* share memory but communicate through message passing. Their verification, in the spirit of Hoare logic, is done by capturing the semantics of message passing in so called "communication events". These parallel algorithms incorporate explicit dynamic process creation in contrast with implicit process creation as done by *cobegin – coend*. Thus we capture the semantics of process creation through the notion of contracts, which are constructed out of communication events.

Biographical sketch

Senior scientist at HP Labs since 1986, about 25 publications on: heuristic search, theorem proving, verification, medical data bases, etc. Current interests: Object-oriented Analysis, Integrated objected-oriented development environments/ tools, Algorithm/ code validation, etc.

VERIFICATION OF SOME PARALLEL ALGORITHMS

Dennis de Champeaux
HP-Labs
1501 Page Mill Rd, 3U
Palo Alto, CA 94304-1181

INTRODUCTION

A new set of arguments about verification emerges due to the spreading of parallel hardware. While sequential programs have been successfully tested, it is not clear whether this practice can be carried over in a parallel setting due to, for instance, the difficulty of reproducing errors when timing plays a crucial role.

We assumw that the software to be verified has an alternative formal specification. In this paper, we develop a technique to formally demonstrate the correctness of parallel algorithms. The correctness of an algorithm is, of course, a relative affair. The technique aims at demonstrating only that an algorithm satisfies its formal specification. There is still no absolute guarantee that when this demonstration succeeds the algorithm embodies our intentions, since both the algorithm and its specification may deviate similarly from the intended behavior.

Since correctness proofs done by "hand" are not to be fully trusted, we have set ourselves the constraint that a verification technique should be easily mechanizable and in particular should not rely on esoteric logics for which deductive machinery is unconsolidated. To be more specific about the significance of this constraint, such a verification system would take as primary input:

- an algorithm - formulated in an agreed upon minimal language,
- a declarative specification - again formulated in an agreed upon language, and
- an optional resource parameter expressing how much resources to spend on proof obligations.

The primary output would be:

- a confirmation of the verification task, or
- a notification which sub task cannot be handled (either due to non-satisfaction of the algorithm to its specification or due to exhaustion of the deductive resource).

Secondary input could consist of a library of previously certified lemmas regarding a particular application domain and/ or tactical control suggestions for the deductive machinery.

We foresee that development of software with such a system would produce very high quality code, especially if different people would develop the algorithmic part and the specification, since making the same mistake twice is quite unlikely.

Due to our mechanizability constraint, we need to give some details of the formalism in which the specification are to be expressed. We have to elucidate as well the procedure to be used for demonstrating that the algorithm satisfies the specification.

First, we deal with the sequential case in section 2 and then illustrate the main ideas on McCarthy's 91-function in section 3. Subsequently, in section 4, we extend the formalism and the technique to cover inter process communication, through send and receive, where the send is asynchronous, and to cover dynamic process creation, through a create-process primitive. The example in section 4 is special in that a compile time analysis can determine which processes will be around - although they are created at run time - and how they interact. An example with an unbounded number of processes and where one cannot analyze beforehand which processes will send messages to each other is treated in section 5.

The examples we treat assume a non-shared memory architecture, where processes interact via message passing. We also assume that a receive operation is blocking when no message is available, while we make no assumption about whether a send is blocking. The non-shared memory assumption is weak since we can represent shared memory as a process that handles read and update messages.

At present our technique cannot be applied to processes that broadcast messages. Neither can we deal with the situation where a message can originate from multiple sources; i.e. our message channels are directional and have exactly two end points.

SYMBOLIC EVALUATION AND THE MONTAGUE CALCULUS

Until now we have been uncommitted on how the satisfaction of an algorithm with respect to its specification is to be established. A standard approach is to identify all reachable positions in an algorithm and attach to every position a most general state description which must be satisfied by each visit to the position. Subsequently, it is to be checked that for each operation in the algorithm it is the case that the pre-conditions of the operation are satisfied by the state $S1$ in which it will be executed and in addition it is to be checked that all the properties in the subsequent state $S2$ are inferable from the initial state $S1$ and the post-condition of the operation. We insist here that all operations are terminating.

A symbolic evaluator assists in constructing the descriptors for all positions. It takes as input an algorithm and its pre- and post-condition description. In case the algorithm contains invocations of operations not belonging to an agreed upon programming language, then the pre- and post- conditions of these operations have to be made available to the symbolic evaluator as well. The descriptions of the other operators are assumed to be built into the symbolic evaluator.

On the basis of all this information available to the symbolic evaluator (ignoring labels and iteration constructs, by assuming only recursion, etc.) it is easy to see how all the descriptions of all positions in the algorithm can be constructed. The proof obligations resulting from the preconditions of the operations in the algorithm can be constructed automatically. Proof obligations in exit positions can be constructed as well.

The next section shows an symbolic evaluator "in action" on the 91-function.

We need a formalism that can deal with incompatible positions - a program variable X can have the value 0 in one position and the value 1 in the successor position - we need a provision to avoid contradiction.

The situation calculus solves this puzzle by adding a situation parameter to keep assertions belonging to different positions apart. Thus, when in a position $S1$, where X would have the value 0, we would execute

```
X <- 1
```

the situation calculus would represent the successor state $S2$ by:

```
Value(X, S1) = 0 & Value(X, S2) = 1,
```

which avoids a contradiction indeed. However, this requires carrying over non-changing aspects from one state to a successor state.

Our specification language - is inspired by intensional logic [Montaque, 1973[4]]. It uses a two level representation, one that keeps track of the assertions (including assertions regarding the past, as is the case in the Situational Calculus), and another one that makes explicit what the extensions are of the entities in a particular situation. Assertions will be formulated in terms of these extensions. To represent a successor position starting from an initial position, we do two things:

- assertions are *added*, where these assertions rely on new extensions for the affected entities, and
- a set of intension - extension pairs, which keeps track of the relevant entities, is *updated* to reflect what is the current set of extensions in the successor position.

Since this approach is not too well known, we give two examples.

Example 1.

Initial position S1 Final position S2

Assume that the action "move C from its position on B to the position on top of D" is executed. The two figures depict the initial and final situation.

The assertions in $S1$:

```
On(b, a) & On(c, b) & Clear(c) & Clear(d).
```

The entities represented by intension - extension pairs in $S1$:

```
((A . a)(B . b)(C . c)(D . d)).
```

The assertions in $S2$:

```
On(b, a) & On(c, b) & Clear(c) & Clear(d) &
Loc(b2) = Loc(b) & Clear(b2) &
Loc(d2) = Loc(d) & On(c2, d2) & Clear(c2).
```

The entities in $S2$:

```
((A . a)(B . b2)(C . c2)(D . d2)).
```

The move of block C induced changing its extension from c into $c2$. Block B is affected by this move, thus it gets a new extension $b2$ as well. Block D is similarly affected. We are still required to carry over unaffected properties of block B, C and D, like their weight, color, etc. Only the extensions of block A does not change its extension suggesting only minimal improvement over the situation calculus. In practice however there will be many more entities that are not influenced by an action.

Example 2.

This example illustrates how we deal with conditionals. We have added position labels ($S0$, $S11$, ...) to facilitate referring to the positions.

```
S0: If Y < 0 Then {S11: X <- Y ; S12: R}
    Else {S21: X <- fac(Y); S22: R}.
```

We sketch what reasonable assertions and intension - extension pairs are at some positions.

The intension - extension pairs and assertions in $S0$:

```
((Y . y)) / Integer(y).
```

In $S11$:

```
((Y . y)) / Integer(y) & y < 0.
```

In $S22$:

```
((Y . y)(X . x)) / Integer(y) & 0 <= y & x = y!.
```

THE 91-FUNCTION

We will show in this section how a symbolic evaluator can generate the proper proof obligations, while employing the Montaque Calculus as the annotation language for the positions. The algorithm is as follows:

```
Function F91(I : integer) : integer;
S0: If 100 < I
          Then { S11: F91 <- I - 10; S12:}
          Else { S21: Z <- F91(I + 11);
                 S22: F91 <- F91(Z); S23: }
```

The position description in $S0$ is:

```
((I . i))/ Integer(i).
```

The output condition to be satisfied at each exit is formulated under the assumption that the extension $f91$ will be returned:

```
((I . i)(F91 . f91))/
{i <= 100 --> f91 = 91} & {100 < i --> f91 = i - 10}.
```

If upon reaching an exit the actual extension of $F91$ is something different, say *foo*, then the proof obligation is in fact:

```
{i <= 100 --> foo = 91} & {100 < i --> foo = i - 10}.
```

We sketch the position descriptions and the proof obligations at the exits for the two branches in the algorithm.

For the branch where the test evaluates to true, we get:

```
S11: ((I . i))/ Integer(i) & {100 < i}.
S12: ((I . i)(F91 . f1))/
     Integer(i) & {100 < i} & {f1 = i - 10}.
```

The proof obligations resulting from the pre-conditions of the operations on this branch (and on the other branch) are ignored since they are trivial. For instance, successful evaluation of the boolean expression requires that the extension of I is a number. This is provable indeed in $S0$.

Since $S12$ is an exit position we have to verify whether the proof obligation:

```
{i <= 100 --> f1 = 91} & {100 < i --> f1 = i - 10}
```

is inferable in $S12$. The two sub-problems - the two implications - are proven easily. The first one because the premise of the implication is false in $S12$; the second one because the consequence of the implication is true in $S12$.

Subsequently, we look into the position descriptions in the other branch:

```
S21: ((I . i))/ Integer(i) & {i <= 100}.
```

We ignore proving the termination of the recursive invocations of $F91$ (which can be done by defining a well ordering on the argument of $F91$ and showing that the two recursive calls have "smaller" inputs than the initial argument I with respect to that well ordering).

```
S22: ((I . i)(Z . z))/
     Integer(i) & {i <= 100} &
     {i <= 89 --> z = 91} &
     {89 < i --> z = i + 1}.
```

The last two conjuncts have been added because we assume that the recursive call is correct and thus yields the proper post-condition, which gets adjusted in our case since the input argument value is actually $i + 11$. Similarly, we obtain for the exit position:

```
S23: ((I . i)(Z . z)(F91 . f2))/
     Integer(i) & {i <= 100} &
     {i <= 89 --> z = 91} &
     {89 < i --> z = i + 1} &
     {z <= 100 --> f2 = 91} &
     {100 < z --> f2 = z - 10}.
```

Since the extension of $F91$ is $f2$, the proof obligation in the exit $S23$ is:

```
{i <= 100 --> f2 = 91} & {100 < i --> f2 = i - 10}.
```

The second sub-problem falls out immediately because the premise of the implication is false in $S23$. The first sub-problem requires more work. By case reasoning and considering the three cases: $i <= 89, 89 < i < 100$ and $i = 100$, we can establish each time that $f2 = 91$ indeed, which we leave as an easy exercise to the reader.

In summary: We employ for each position a two level characterization consisting of:

- the significant entities visible in the position with their extension, and
- a set of assertions that describe the features of the current extensions and possibly also of previous extensions.

An action leads to the introduction of new extensions for those entities that are affected while new properties and relationships between old and new extensions are added to the set of assertions. An if-then-else construct forces the symbolic evaluator to pursue two branches of positions, one branch in which the test evaluates to true and the other branch where the test evaluates to false. This shows that real examples still play a role in verifying algorithms, because they demonstrate that all branches are hit. If we would succeed in proving that a boolean in a test holds in its initial position then we would have demonstrates that the else-part is dead code.

In the next section, we will extend this formalism in order to deal with inter process communication and process creation.

THE PARALLEL SET PARTITION ALGORITHM

A parallel algorithm consists of a family of sequential modules. For verification, we want to apply divide and conquer, i.e. our technique should allow dealing with one module at the time. Inter module communication is to be verified only for those modules which actually interact. Global properties are to be inferable from the results of the separate verification activities, and thus they should not rely on the innards of the modules.

The example to be presented, contains invocations of send's and receive's for inter process communication as well as invocations for process creation. We address first the question of how to ignore the process at the other side of a communication event. Phrased alternatively, we deal with the question of how to formulate the pre- and post-condition of a communication primitive without relying on the "inside" of a partner process.

At first sight, a receive, say executed by process P, does not have any pre-condition. There is no property of the entities at the position preceding a receive that can affect the success or failure of the receive operation. Similarly for the post-condition of a send operation. To appreciate these features we consider a process Q that starts with a receive and ends with a send operation:

```
S0: Receive ( ... );
...
Sn: Send ( ... );
Sexit:
```

This module does *not* have a pre-condition nor a post-condition. The state of the "world" in $S0$ as well as in *Sexit* is immaterial.

At the same time, it is clear that something must hold for the *Receive* and *Send* to make sense. Although the process Q dies after the *Send*, this operation fulfilled some purpose.

The semantics of these operations are captured by observing that the processes cooperate and as such have to obey contracts. Each process can be seen as having to fulfill *obligations* and in return can have *expectations* regarding the other processes. Thus we adjoin to each sequential module a description of the communications that it has with other modules. To be more precise, we will add to the state description of every position in a module a characterization of the communication events that should still occur in some future before the execution of the module terminates.

At this point the position description consists of the present intension/ extension set and of features of the future - communication events. In the next section, we will adjoin to a position features of the past history that explains how a position has been reached.

Since the modules are sequential, the communication events form a linear sequence. To describe the structure of this sequence, we will exploit the constraint that we have ruled out, for now, broadcasting and its dual: a channel that can receive messages from multiple sources.

Each element in the linear sequence of communication events contains several pieces of information. We will only sketch here the "inside" of a communication event description. A unique extension refers to the value that is transferred. Another field describes the properties of this value. It turns out that the property description of the value transferred may depend on the extensions of entities that are private to the sender or even to third party processes that the sender knows about. In case the recipient processes has to know about the intension - extension descriptions of these auxiliary entities, they will be part of a communication event descriptor as well.

When a module does not have communication operations in iterative constructs or inside recursive constructs, we could spell out its sequence of communication events once and for all. However in general, as in our example, the sequence of communications is a function of the data that is being processed. Consequently, we need constructs for generating communication event descriptions. Below, in the context of our example, we will go into more details.

In summary for now: a module is characterized not only by pre-conditions but also by its contracts which describe the module's communication events. Each position in the module has in addition to a state description a component that describes the future communication events.

We have ***not*** incorporated post-conditions in the characterization of a module. Modules do not return values, because there is nobody waiting for them. The pre-conditions have been retained to permit data to be passed on during process creation from the creator to arguments of a procedure that will be started up.

We now have enough material to discuss the semantics of process creation, ignoring the pre-conditions induced by system mandated privileges for the parent process.

If a module has pre-conditions regarding properties of data that comes from the parent process at creation time, then we have pre-conditions for the instruction which creates the process instance of that module. The post-condition of the creation instruction is simply the unchanged state in which the instruction started execution. We still have to account for the contracts associated with the module. To facilitate their explanation, we assume for the moment that all subsidiary processes are generated by one master, initial process.

Creation of the first subsidiary process generates the foundation for the above introduced contracts. The communication event descriptions of the module may induce ***dualized*** descriptions for the creator.

For instance, if the new process has an expectation to receive a value from the parent then we infer a communication obligation from the parent towards the new process. Symbolic evaluation of the remainder of the parent module needs to resolve such a communication obligation which entails that induced pre-conditions, as specified in the communication event description, must be satisfied in the position where the relevant *Send* is being invoked.

Communication events in the module's contracts towards other processes are put on hold. These have to be matched against similar descriptions of other subsidiary processes that are created by the master process.

For example, assume that the initial process creates the subsidiary processes A and B, while the contract of A contains a communication event involving process B. This communication event from A's perspective must be reconciled by the description from B's perspective.

In this section, we can equate a module and its activation in a process. We have to be more careful in the next section where a module can be activated arbitrarily often.

The example we are going to treat is attributed to Dijkstra and is used in [Barringer, 1985 [1]] to illustrate techniques developed in [Lamport, 1980 [2]] and [OG, 1976 [5]]. They use two subsidiary processes in a shared memory context with shared variables for synchronization. Our algorithm employs three subsidiary processes - creating a symmetric situation - that communicate by message passing. They do not deal explicitly with sub-process creation - their *cobegin – coend* is doing the job. We will deal with the semantics of a create-process primitive.

The task is to partition elements in two non-empty bags of integers, $bagS$ and $bagL$ such that:
- the size of the bags at the end is the same as at the beginning,

```
|bagSin| = |bagSout|, |bagLin| = |bagLout|
```

- no elements are lost or are created in the meantime,

```
bagSin U bagLin = bagSout U bagLout
```

(U stands for bag union, = stands for bag-equality);
- at the end we have that the maximum element in *bagS* is less or equal than the minimum element in *bagL*,

```
max(bagSout) <= min(bagLout)
```

- the bags give away at most one element at the time (this constraint excludes an implementation where the elements of both bags are written into a sequence, the sequence is sorted the lower part is written back in *bagS* and the upper part written back into *bagL*.)

Example:

Input:

```
bagS =  (1 9 9)
bagL =  (6 3 5 8)
```

Output:

```
bagS =  (1 3 5)
bagL =  (9 9 6 8)
```

An iterative sequential solution might be something like:

```
ready <- false
While Not(ready) Do
    [p <- max(bagS);
     q <- min(bagL);
     If q < p Then
         {bagS <- bagS - p + q;
          bagL <- bagL - q + p}
     Else ready <- true]
```

or recursively:

```
sift(bagS, bagL);
     [p <- max(bagS);
      q <- min(bagL);
      Return
         {If q < p Then sift(bagS - p + q, bagL - q + p)
          Else <bagS, bagL>}]
```

In our parallel solution, we will have a *Main* program (part of a larger module) which "owns" the initial bags and whose only activities is to create the subsidiary processes that will do the work. There are two subsidiary processes responsible respectively for *bagS* and *bagL*, and a third comparator process.

Without further ado we give the code of the modules:

The code of *Main*:

```
...
Create-proc(S, S_Proc()); S/L/C are the process names
Create-proc(L, L_Proc()); S-Proc/ etc are the procedures
                        ; executed
Create-proc(C, C_Proc()); in the respective processes
Send(S, bagS);
Send(L, bagL);
Receive(S, bagS);
Receive(L, bagL);
...
```

The module S that is responsible for the *bagS* executes the procedure *S_Proc*:

```
S_Proc();
     [Receive(Main, bagS);
      out <- S_func(bagS);
      Send(Main, out)]
```

The recursive support function employed inside *S_Proc*:

```
S_func(bagS);
     [p <- max(bagS);
      Send(C, p);      send the max element to the comparator
      Receive(C, p2);  get the reply back
      Return(If p = p2 Then bagS Else S_func(bagS - p + p2))]
```

The module C which compares and switches elements executes the recursive procedure *C_Proc*:

```
C_Proc();
     [Receive(S, maxS);
      Receive(L, minL);
      If maxS <= minL
      Then {Send(S, maxS);     return the same elements
            Send(L, minL)}     and terminate
      Else {Send(L, maxS);     return the exchanged elements
            Send(S, minL);
            C_Proc()}]         and recurse
```

We leave the description of the module L and of its support function *L_func* to the reader.

Main start:

Global:

```
G_BagS = bagS; G_BagL = bagL
```

Bindings:

```
((bagS . bs_in)(bagL . bl_in))
```

State:

```
~Empty_bagi(bs_in) & ~Empty_bagi(bl_in)
```

G_BagS and *G_BagL* are global names for the bags. This allows the modules to identify what they are referring in the descriptions of communication events.

It is worth noting that there is no contract description above. This is correct because at the point where *Main* starts executing it is not yet known that the solution will involve other processes.

To shorten the descriptions below, we will make use of an abbreviation, which is quantified over *so* and *lo*. It expresses that the bag-pair $< so, lo >$ is a permutation of the input bag-pair $< bs\_in, bl\_in >$:

```
preserve_content&size(so, lo) <-->
   [ Union_bagi(so, lo) = Union_bagi(bs_in, bl_in) &
     Size(so) = Size(bs_in) &
     Size(lo) = Size(bl_in) ]
```

Thus we can formulate the exit as:

Main exit:

Global:

```
G_BagS = bagS; G_BagL = bagL
```

Bindings:

```
((bagS . bs_out)(bagL . bl_out))
```

State:

```
preserve_content&size(bs_out, bl_out) &
Max_bagi(bs_out) <= Min_bagi(bl_out).
```

The lack of contracts is significant. There are no outstanding expectations or obligations. And in case symbolic evaluation would reach the end with an outstanding obligation, we would know that the implementation would be incorrect.

The annotations given above make use of not yet defined functions, like *Size*, and a non-defined predicate *Empty_bagi*. Their meaning becomes important when proof obligations are to be checked. At that time, axioms, definitions and lemmas from the domain of bags have to be given to the deductive machinery. For example, lemmas regarding the commutativity and associativity of *Union_bagi* may be relevant, or a lemma like (we assume the variable x to be of type bag):

```
(x) { Size(x) = 0 <--> Empty_bagi(x) }.
```

The module S has empty pre- and post-conditions since it receives its input from *Main* and sends the results back to *Main*. It has no contract at the exit. At the start of S, we do have a contract. It consists of three parts:
(1) an explicit description of the incoming message from *Main*,
(2) a description of a series of exchanges between S and C, and
(3) an explicit description of the outgoing message to *Main*.

We describe the format of (1) and (3) first. Each message corresponds with a communication event (= ce) that we notate in a tuple format with the following five fields:

- whether the message is incoming or outgoing as indicated by *expec* respectively *oblig*;
- name of the partner process;
- global and local name of the *value* to be transferred; the global name is used by all processes in order to identify a particular entity, the local name is used to describe the current extension that is exchanged in the ce; we consider the inclusion of the transfer of a possession token in the ce tuple; if so we could express whether the exchange is for read-only purposes or alternatively whether the receiver has update privileges;
- global and local names of other entities which are required to characterize the value of the entity being transferred;
- assertions regarding the transferred value, possibly in relationship with values of other global entities.

A send-operation derives its proof obligation from the assertions in the last field. The ce (1) that describes the first exchange between *Main* and S (from the perspective of S):

```
{ expec/
  *parent*/
  (G_BagS . bs_in)/
  (G_BagL . bl_in)/
  ~Empty_bagi(bs_in) }
```

Similarly, the ce (3), in which the bag is returned from S to *Main*, is:

```
{ oblig/
  *parent*/
  (G_BagS . bs_out)/
  (G_BagL . bl_out) (G_MaxS . pn) (G_MinL . qn)/
        preserve_content&size(bs_out, bl_out) &
        pn = Max_bagi(bs_out) &
        qn = Min_bagi(bl_out) &
        pn <= qn }.
```

The process L owns *G_bagL*. In order for the process S to make sense of the values it receives from process C, S needs to know about the most recent version that L has according to process C. Similarly, the

correctness of the *Main* process depends crucially on the properties of the values of *G_MaxS* and *G_MinL* as reported by process *C* and as passed on to *Main* by process *S* as well as by process *L*.

This leaves us with the formulation of the ce (2), the series of exchanges between *S* and *C*. These exchanges depend on the content of the bags. In the special case that initially the maximum of *bs_in* is already less or equal than the minimum of *bl_in*, then we will have only one series of exchanges between *S* and *C* (and likewise between *L* and *C*). Otherwise, we will have more exchanges.

The function *S_C_CEs* below describes the communication events between *S* and *C* from the perspective of *S*. Auxiliary functions are introduced to break down the complexity. The function *SC_Send* describes a send ce from *S* towards *C*. The associative function || stands for the concatenation of ce's. The function *SC_Receive_and_continue* describes the ce which originates in *C* and is received by *S*, possibly followed by a recursive invocation of *S_C_CEs*. The arguments have the following interpretation:
sn = most recent value (known) of the global entity *G_BagS*
so = previous value, provided $sn \neq bs\_in$
ln = most recent value (known) of the global entity *G_BagL*
lo = previous value, provided $ln \neq bl\_in$
pn = *Max_bagi(sn)*
po = *Max_bagi(so)*, if *so* is defined
qn = *Min_bagi(ln)*
qo = *Min_bagi(lo)*, if *lo* is defined

Since L owns *G_BagL*, *S* has (in general, when $sn \neq bs\_in$) more up-to-date info about *G_BagS* than about *G_BagL*. Thus while *S* knows about *sn* and about the previous version *so*, it will only know about the less recent version *lo* of *G_BagL*.

This gives us:

```
S_C_CEs(pn, po, sn, so, qo, lo) =
        SC_Send(pn, po, sn, so, qo, lo) ||
        SC_Receive_and_continue(pn, po, sn, so, qo, lo).
```

In the following descriptor, we see that, for instance, *G_MaxS* has two extensions: *pn* and *po*. The most recent one is *pn*, while *po* is the preceding value. However, as an exception *po* as well as *so* are undefined when *SC_Send* describes the first ce between *S* and *C*. The first communication between *S* and *C* is taken care of by the first part of the disjunction in the body *SC_Send*:

```
SC_Send(pn, po, sn, so, qo, lo) =
{oblig/
   </
   (G_MaxS . pn . po)/
   (G_BagS . sn . so) (G_BagL . lo) (G_Q . %qo)/
   pn = Max_bagi(sn) &
   [ ( sn = bs_in &
       lo = bl_in ) or
     ( sn = so - po + qo &
       preserve_content&size(so, lo) ) ] }.
```

Subsequently, we describe the reply from *C* towards *S*. In the body, we encounter an existential quantifier to deal with new extensions that are not functionally dependent on the given arguments. The auxiliary function *S_Receive* deals with the value that is communicated back from *C*. Dependent on whether *S* receives the originally transmitted pn, we will terminate the ce sequence with *FIN* or the sequence will be extended via a recursive invocation of *S_C_CEs*:

```
SC_Receive_and_continue(pn, po, sn, so, qo, lo) = out <-->
  (E qn ln xs)
     out = [S_Receive(pn, po, sn, so, qo, lo, qn, ln, xs) ||
            (if ( xs = pn ) then FIN else
             S_C_CEs(Max_bagi(sn - pn + qn), pn,
                     sn - pn + qn, sn, qn, ln))].
```

The auxiliary function *S_Receive* is quite straightforward. S will know not only that $< so, lo >$ is a permutation of $< bs\_in, bl\_in >$, but also a permutation of $< sn, ln >$.! This key fact, which S needs to pass on ultimately to its parent, will have to be proven during the symbolic evaluation of C, see below. Process S is also informed about the newer version ln of G_BagL:

```
S_Receive(pn, po, sn, so, qo, lo, qn, ln, xs) =
{expec/
   S/
   (G_ToS . xs)/
   (G_BagL . ln . lo) (G_MinL . qn . qo)/
   qn = Min_bagi(ln) &
   ( [xs = pn & pn <= qn] or
     [xs = qn & ~(pn <= qn)] ) &
   ( sn = bs_in &
     ln = bl_in ) or
   ( ln = lo - qo + po &
     preserve_content&size(sn, ln) ) }
```

We leave a similar description for the ce's that take place between process L and C from L's perspective to the reader. Instead, we formulate the ce's from the perspective of C. Auxiliary functions are used again. The top level function describes two incoming ce's and, depending on the values pn and qn that are received, will describe either replies of two terminating ce's or two ce's followed by a recursion:

```
C_S_L_CE(pn, po, sn, so, qn, qo, ln, lo) =
    CS_Receive(pn, po, sn, so, qo, lo) ||
    CL_Receive(po, so, qn, qo, ln, lo) ||
    if pn <= qn then
       CS_Send&Halt(pn, po, sn, qn, qo, ln, lo) ||
       CL_Send&Halt(pn, po, sn, so, qn, qo, ln)
    else
       CS_Send&Continue(pn, po, sn, qn, qo, ln, lo) ||
       CL_Send&Continue(pn, po, sn, so, qn, qo, ln) ||
       C_S_L_CE(Max_bagi(sn - pn + qn), pn, sn - pn + qn, sn,
                Min_bagi(ln - qn + pn), qn, ln - qn + pn, ln).
```

The next two auxiliary functions are responsible for the incoming ce's from S respectively L:

```
CS_Receive(pn, po, sn, so, qo, lo) =
{expec/
 S/
 (G_MaxS . pn . po)/
 (G_BagS . sn . so) (G_BagL . lo) (G_MinL . qo)/
 pn = Max_bagi(sn) &
 [ ( sn = bs_in &
     lo = bl_in ) or
   ( sn = so - po + qo &
     preserve_content&size(so, lo) ) ] }

CL_Receive(po, so, qn, qo, ln, lo) =
{expec/
 L/
 (G_MinL . qn . qo)/
 (G_BagL . ln . lo) (G_BagS . so) (G_MaxS . po)/
 qn = Min_bagi(ln) &
 [ ( so = bs_in &
     ln = bl_in ) or
```

```
    ( ln = lo - qo + po &
      preserve_content&size(so, lo) ) ] }
```

The last four auxiliary functions take care of the replies to S respectively L. Although at input time C knows only that $< so, lo >$ is a permutation of $< bs\_in, bs\_out >$, it guarantees that $< sn, ln >$ is a permutation too:

```
CS_Send&Halt(pn, po, sn, qn, qo, ln, lo) =
  {oblig/
   S/
   (G_ToS . pn)/
   (G_BagL . ln . lo) (G_MinL . qn . qo)/
   pn <= qn &
   qn = Min_bagi(ln) &
   ( sn = bs_in &
     ln = bl_in ) or
   ( ln = lo - qo + po &
     preserve_content&size(sn, ln) )}

CL_Send&Halt(pn, po, sn, so, qn, qo, ln) =
  {oblig/
   L/
   (G_ToL . qn)/
   (G_BagS . sn . so) (G_MaxS . pn . po)/
   pn <= qn &
   pn = Max_bagi(sn) &
   ( sn = bs_in &
     ln = bl_in ) or
   ( sn = so - qo + po &
     preserve_content&size(sn, ln) )}

CS_Send&Continue(pn, po, sn, qn, qo, ln, lo) =
  {oblig/
   S/
   (G_ToS . qn)/
   (G_BagL . ln . lo) (G_MinL . qn . qo)/
   ~(pn <= qn) &
   qn = Min_bagi(ln) &
   ( sn = bs_in &
     ln = bl_in ) or
   ( ln = lo - qo + po &
     preserve_content&size(sn, ln) )}

CL_Send&Continue(pn, po, sn, so, qn, qo, ln) =
  {oblig/
   L/
   (G_ToL . pn)/
   (G_BagS . sn . so) (G_MaxS . pn . po)/
   ~(pn <= qn) &
   pn = Max_bagi(sn) &
   ( sn = bs_in &
     ln = bl_in ) or
   ( sn = so - qo + po &
     preserve_content&size(sn, ln) )}
```

As mentioned above, some global entities have more than one extension. The C process receives different information about the extension of G_BagS from S and from L. S gives: $(sn.so)$ and L gives only so. The overlap of extensions allows the C process to inform the L process in its reply about the more recent extension sn. And similarly S is informed about the more recent extension ln of G_BagL.

We are now able to do correctness proofs.

Lemma 1. The description of the ce's between S and C from the perspective of S respectively C are compatible (and similarly for the ce's between L and C).

Proof. The ce's inside *SC_Send* and *CS_Receive* are obviously compatible. Similarly, *S_Receive* deals in a condensed form - through a disjunction - with the corresponding two cases that are described in *CS_Send&Halt* and *CS_Send&Continue*. □

Lemma 2. The sequence of ce's exchanged between S and C is identical (and similarly for the L - C communication).

Proof. We use here the notation {*proc*1 *proc*2} to express a ce from *proc*1 to *proc*2, while * stands for the closure operation. From the perspective of S, we have (omitting the communications with the parent):

```
[{S C} || {C S} ||]*
```

From the perspective of C, we have :

```
[{S C} || {(L C} || {C S} || {C L} ||]*
```

projecting out the ce's regarding L gives the same sequence.

The termination conditions for both perspectives are identical as well. □

Lemma 3. The code of *S_func* is correct with respect to:

Pre-condition:

Globals:

```
G_BagS = bagS,
```

Bindings:

```
((bagS . sn . so))
((G_BagL . lo)(G_MinL . qo)(G_MaxS . po))
```

Contract:

```
S_C_CEs(pn, po, sn, so, qo, lo)  || ce (3)
```

Remark. We have simplified the notation here. The contract refers to *pn* which is in fact a future extension of the variable *p* in the body of *S_func*. To be precise, we would have to replace *pn* by something like "The extension of *p* when the send occurs".

State:

```
( (sn = bs_in & lo = bl_in) or
  (sn = so - po + qo &
   preserve_content&size(so, lo)) ) &
~Empty_bagi(sn).
```

Post-condition:

Globals:

```
G_BagS = RETURN,
```

Bindings:

```
((RETURN . bs_out))
((G_BagL . bl_out)(G_MinL . qn)(G_MaxS . pn))
```

Contract:

```
ce (3)
```

State:

```
preserve_content&size(bs_out, bl_out) &
pn = Max_bagi(bs_out) &
qn = Min_bagi(bl_out) &
pn <= qn.
```

(and similarly for *L_Proc* the procedure executed by L).

Proof. The assignment statement in *S_func* creates a new extension - to simplify matters say *pn* - of *G_MaxS* and adds the assertion $pn = Max\_bagi(sn)$. The subsequent *Send* operation consumes a ce from the sequence of ce's that is obtained by the expansion of:
$S\_C\_CEs(pn, po, sn, so, qo, lo)$.
The requirements of that ce are satisfied in the state in which the *Send* is executed. The subsequent *Receive* operation consumes again a ce and the state description is expanded with what the ce promises about the value returned. In particular, the variable $p2$ obtains a new extension xs which has all the proper features. Symbolic execution of the conditional statement exploits the disjunction that is added to the state description in the preceding *Receive* operation. We have now two cases.

In case: $p = p2$. We are left with the contract ce (3) as demanded by the post-condition. The other assertions in the post-conditions are satisfied because the actual bindings as reported by *S_Receive* are *sn*, *ln*, ... instead of *bs_out*, *bl_out* ...

In case: $p \neq p2$. We recurse and since we update the input argument in the proper way, we satisfy the pre-conditions of *S_func*. □

Lemma 4. The code of *S_Proc* is correct with respect to:
Pre-condition:
Globals:

```
G_BagS = bagS,
```

Contract:

```
ce (1) || S_C_CEs(pn, po, sn, so, qo, lo) || ce (3)
```

Post-condition: NIL
Proof. The symbolic execution of the *Receive* from the parent is justified by ce (1). The resulting state allows the call of *S_func*, because of the first term of the disjunction inside *SC_Send*. The post-condition of *S_func* guarantees the requirements as demanded by ce (3). Subsequent symbolic execution of the reply to the parent fulfills the contract. □

Lemma 5. The code of C is correct with respect to:
Pre-condition:
Globals:

```
G_MaxS = maxS, G_MinL = minL
```

Contract:

```
C_S_L_CE(pn, po, sn, so, qn, qo, ln, lo)
```

Post-condition: NIL
Proof. The ce's for both receive's are accounted for in *C_S_L_CE*. The branching required by the conditional statement occurs in *C_S_L_CE* as well. The only point of interest are the promises made in the replies to S and L. In particular, we have to infer:

```
preserve_content&size(sn, ln).
```

The *receive* from S has produced:

```
( sn = bs_in &
  lo = bl_in ) or
( sn = so - po + qo &
  preserve_content&size(so, lo) )
```

While the *receive* from L has given:

```
( so = bs_in &
  ln = bl_in ) or
( ln = lo - qo + po &
  preserve_content&size(so, lo) )
```

We now have either that
$sn = bs\_in$ and $ln = bl\_in$,
which makes the task trivial, or that
$sn = so - po + qo$, $ln = lo - qo + po$ and $preserve\_content\&size(so, lo)$.
Our task is reduced now to showing certain properties of bags. For example, it is to be shown that removing an element from a bag and adding an element doesn't change the size of the original bag, etc. □

Lemma 6. The description of the ce's between S and $Main$ from the perspective of S respectively $Main$ are compatible (and similarly for the ce's between L and $Main$).

Proof. The contract for $Main$ is obtained - as a consequence of the semantics of the $Create\_proc$ operations - by the dualization of those ce's in the contract in S (and L) that are devoted to $Main$. Thus the perspective from $Main$ is equal to that of S by construction. □

Lemma 7. The contract for $Main$, induced by the creation of S and L, supports its post-condition.

Proof. The "dual" of ce (3) for S and L produces the properties as required by the post-condition of $Main$. □

Lemma 8. $Main$ terminates.

Proof. This property boils down to the termination of C. Each iteration of C the number of swappable pairs decreases by one. The maximum of iterations is bounded by: $minimum(size(bagS), bagL)$. □

Theorem. The parallel implementation of $Main$ with explicit process creation and ipc is correct.

THE PARALLEL OLYMPIC TORCH

The example in the previous section was limited in that the set of processes which could come into being was fixed. In addition, the communications between these processes could be analyzed easily because processes were connected with fixed channels. The example we will discuss in this section have an unbounded number of processes all of which are instances of the same module. The communication structure is dynamic since processes send messages into mailboxes and they have no idea which process receives them.

The Olympic Torch was inspired by an example presented by Ehud Shapiro in the context of Concurrent Prolog.

The task is simple: given the root of a binary tree, apply the procedure *proc* to the leaves in a sequential order from left to right. The procedure *proc* could, for instance, be a print operation.

An obvious ***sequential*** solution involves a recursive procedure *Proc_tree* which can be coded - assuming the usual utility operations *Leaf*, *Left* and *Right* - as:

```
Proc_tree (arg)
     If Leaf (arg)
     Then proc (arg)
     Else [ Proc_tree (Left (arg));
            Proc_tree (Right (arg)) ]
```

A parallel solution: *Create_process* takes here only one argument, the expression to be evaluated/ executed by the new process; we don't need to provide process names here since all ipc goes via mail boxes, which we *do* pass along; observe as well that we use mail-box versions of *Send* and *Receive* which are respectively *Send_m* and *Receive_m*

```
Visit_leaves_in_left_to_right_order(arg);
  mailbox_left <- Make_mailbox;
  mailbox_right <- Make_mailbox;
  Create_process (Proc_tree2 (mailbox_left, arg,
```

```
                                mailbox_right));
  Send_m (mailbox_left, "torch");
  Receive_m (mailbox_right, out);
```

where the procedure *Proc_tree2* is:

```
Proc_tree2 (mailbox_left, arg, mailbox_right);
  If leaf (arg)
  Then [ Receive_m (mailbox_left, torch);
         proc (arg);
         Send_m (mailbox_right, torch) ]
  Else [ mxm <- Make_mailbox;
         Create_process (Proc_tree2 (mailbox_left,
                                     Left (arg), mxm));
         Create_process (Proc_tree2 (mxm, Right (arg),
                                     mailbox_right)) ]
```

While the order of the recursive invocations in the sequential algorithm is crucial for the correctness, the order in which the subsidiary processes are created is immaterial.

The reason why this solution is correct is not so obvious. Will we visit all the leaves of arg? Will we visit them in left-to-right order? These properties are unusual in that they are not a property of the final state, they do not belong to the post condition. Instead, they refer to the run time behavior, or alternatively control flow, of the algorithm. As such, these features resemble time/ space complexity features, which are also not immediately expressible in terms of predicates on the final state.

Let's first see how proofs for the sequential case may be done:

- Will we visit all the leaves of arg?

case 1: arg is a leaf. We will visit arg indeed, because proc will be applied on arg.

case 2: arg is not a leaf. The tree hanging of arg consists of two sub-trees. By induction we know that all the leaves of these sub-trees are visited, etc.

- Will we visit all the leaves in a left-to-right order?

case 1: arg is a leaf. Trivial.

case 2: arg is not a leaf. By induction, the leaves are visited in both sub-trees correctly and indeed the call to the left sub-tree precedes the call to the right sub-tree.

There is a way to reduce this run time behavior analysis to state space reasoning by letting states describe how they came into being.

We capture the event of visiting a leaf node by appending a node description to an event list. The node description is gradually build up at each recursive invocation of *proc_tree*. A left branch of a non-terminal node is marked by a 0, a right branch by a 1. The node description consists of concatenating all link labels on the path from the leaf to the root, starting at the root. We can interpret the list of 0-1 digits ($x1x2...xn$) as corresponding with a binary number in the range [0,1) represented by: $0.x1x2x3...xn$. The left-to-right visiting of the leaves is equivalent to the assertion that the sequence of numbers on the *event_list* is increasing (the *event_list* is manipulated by an operation push which adds elements to the end of the list). The extended algorithm is:

```
Proc_tree3 (arg, event_num)
     If leaf (arg)
     Then [ proc (arg);
            Push (event_num, event_list) ]
     Else [ Proc_tree3 (Left (arg), Append (event_num, 0));
            Proc_tree3 (Right (arg), Append (event_num, 1)) ].
```

Before *Proc_tree3* is invoked, we assume that *event_list* has been initialized:

```
event_list <- {0}.
```

An initial invocation of *Proc_tree3* would be of the form:

```
Proc_tree3 (arg, |1|).
```

Informal correctness proof:

Input:

```
event_num > last(event_list_in)
```

Output:

```
event_list_out = event_list_in /
                 (description of leave nodes in
                  left-to-right order of arg)
```

where / stands for a concatenation operation.

This is true for the base case when arg is a leaf.

Let's look at the case when arg is not a leaf. We need to show that the additions to the *event_list* conform to the *event_list* properties. This is equivalent to showing that all the new additions are greater than the elements on *event_list_in* and that the new additions are correctly ordered among each other.

1) all the leaves in arg have description values greater or equal than *event_num*; thus they are ordered with respect to the elements that occur on *event_list_in* (since we know that *event_num* > *last*(*event_list*) = the maximum element on the *event_list*).

2) the leaves in the right sub-tree are greater than the leaves in the left sub-tree; thus the preconditions of the second call are satisfied. By induction, we know that the two recursive calls add the proper sub-sequences to *event_list*.

Now we extend the parallel solution with similar virtual code to capture the events of leaf processing. There are two problems with this solution because of the global variable *event_list*. First our processes have been conceptualized as not sharing memory, which makes access to *event_list* as presented in the code illegal. Second, since multiple processes will need to have access to *event_list*, we should worry about multiple processes competing for access to *event_list*. Both of these problems can be dealt with by introducing a custodian process for *event_list*. However, we will leave the situation as is because *event_list* is a virtual construct and access to *event_list* is inherently sequential, as will be shown below.

Parallel solution - with *event_list* extension:

```
Visit_leaves_in_left_to_right_order2(arg);
  mailbox_left  <- Make_mailbox;
  mailbox_right <- Make_mailbox;
  event_list <- {0};
  Create_process (Proc_tree4 (mailbox_left, arg, |1|,
                              mailbox_right));
  Send_m (mailbox_left, "torch");
  Receive_m (mailbox_right, out),
```

where the procedure *Proc_tree4* is:

```
Proc_tree4 (mailbox_left, arg, event_num, mailbox_right);
  If Leaf (arg)
  Then [ Receive_m (mailbox_left, torch);
         proc (arg);
         Push (event_num, event_list);
         Send_m (mailbox_right, torch) ]
  Else [ mxm <- Make_mailbox;
         Create_process
           (Proc_tree4 (mailbox_left,
                        Left (arg),
                        Append (event_num, 0),
                        mxm));
         Create_process
```

```
(Proc_tree4 (mxm,
 Right (arg),
 Append (event_num, 1),
 mailbox_right)) ]
```

The claim, of course, is that we have again the same elements on the *event_list* as in the sequential case.

The main categories of the state descriptions in the annotation language until now are:

– set of assertions that apply to the current state and possibly referring to previous states via preceding still contributing extensions;

– list of intension - extension pairs;

– contract, i.e. future communications with other processes; a distinction is made between expectations and obligations in order to cater for respectively receives and sends.

To be added:

a generic *event_list* onto which a generic operator can add event descriptions (to be used, e.g., for complexity proofs/ process features, etc.); this *event_list* consist of two parts:

- events that have occurred already, and
- events that will occur in some future.

In our application, the elements on the *event_list* are simple atomic events. It is conceivable however that another application needs non determinism, i.e. where an element of the *event_list* stands for a set of unordered atomic events.

We annotate *Proc_tree4* at the entrance with the future event obligations of visiting all the leaf nodes. While, we annotate the exit with having accomplished all these obligations. In the base case, *Proc_tree4* will have done the work itself; alternatively the work will have been done by newly created processes.

The only worry we have is that the delegation involves initially two sub-processes. Thus we have to ascertain that these two sub-processes coordinate well.

Lets look at the contract of *Proc_tree4*:

Social-contract:

```
< { expec/
    "torch"/
    read(mxl)/
    ((EVENT_LIST . event_list_in)(MXL . mxl)(MXR . mxr))/
    >PM<
    ( last(event_list_in) < event_num )
    >PM< }
  { oblig/
    "torch"/
    send(mxr)/
    ((EVENT_LIST . event_list-out)(MXL . mxl)(MXR . mxr))/
    >PM<
    event_list-out = event_list_in/Z &
    Z = Events-of-visiting(node, event_num)
    ; Z contains the descriptions of the leaves of node
    >PM< } >
```

We have to define the descriptor *Events_of_visiting*:

```
Events_of_visiting (node, event_num) = Z <-->
   [ leaf (node) --> Z = {event_num} ] &
   [ ~leaf (node) -->
     Z = Events_of_visiting (left (node), event_num|0) /
         Events_of_visiting (right (node), event_num|1) ]
```

If we can show that *Proc_tree4* supports this contract then it is easy to show that the *Main* program visits all the leaves of *arg* in the required way. Because the *event_list_out* has the description of the leaves in the left-to-right order. Thus the remaining task is to show that *Proc_tree4* lives up to the contract.

The base case, when arg is a leaf is obvious, since *Proc_tree4* itself updates the *event_list*. However in the non-leaf case, we encounter delegation. By induction, we get the following two additions to the *event_list*:

```
Events_of_visiting (left (node), event_num|0),
Events_of_visiting (right (node), event_num|1).
```

Thus we have the guarantee that the proper set of elements will be added to *event_list*. Still, it is possible that the two sequences are interleaved. We see that the additions, that are the consequence of the right sub-tree, happen *after* the receive from *mxm* is done. Which is preceded by the send into *mxm*, which is preceded by the addition of the left sub-tree. Transitivity of precede settles the argument. □

The *event_list* that we introduced will be treated like the ladder that is to be thrown away after we gained the insight by climbing it. There is no connection between *event_list* and the procedure *proc* that gets executed when a leaf is encountered. The manipulations on the *event_list* serve only the purpose of demonstrating the sequential left-to-right control flow of both sequential and parallel algorithms. Thus by eliminating *event_list*, we will preserve the properties that we verified.

DISCUSSION AND COMPARISON WITH OTHER WORK

The set partition problem has been dealt with before. In [Barringer, 1985 [1]], we find sketches of three treatments. Two of these illustrate how to deal with processes that have access to shared variables. The third one is closer to our treatment, since message passing a la CSP is exploited.

We consider the message passing approach superior. The message passing primitives themselves have to be implemented with semaphores, i.e. with shared variables and so one cannot really avoid shared variables complexities. However, their correctness has to be shown only once and for all and is independent of idiosyncrasies of application programs.

The third approach, with message passing, illustrates work done by [Levin, 1980 [3]]. The technique is summarized in [Barringer, 1985 [1]]:

```
Basically, a proof of 'weak' correctness (total correctness
without deadlock freedom proofs) consists of sequential proofs
(considered in isolation), satisfaction proofs and non-interference
non-interference proofs.
```

This sounds quite similar with what we did in section 4, with the exception of the non-interference proofs. We did pass information along concerning global entities but each global entity had a unique custodian and interference was consequently not an issue.

As far as we know, we are the first to have formulated the semantics of explicit process creation through so called contracts which describe a process in terms of the communication expectations and obligations.

The treatment of the Olympic Torch is as far as we know a first as well, although the proof relies on the global entity *event_list* and as such is somewhat unsatisfactory. At the other hand, we deal in this algorithm with arbitrary many processes.

CONCLUSION AND SUMMARY

We have developed a technique for proving the correctness of parallel algorithms which use message passing for inter-module communications. A calculus is given for describing inter-module communication events, which lead to a notion of contracts. These form the foundation for the semantics of a sub-process creation primitive. A correctness proof first shows that each module (and each process) fulfills its contract and then shows that mutual views of communicating processes are compatible.

The method is explained in more detail through application on two examples. The first example deals only with a limited number of processes while the second deals with an unbounded number of sub-processes.

Acknowledgement

Alex Stepanov's careful reading of a draft version modified the presentation throughout.

References

[1] BARRINGER, H., A Survey of Verification Techniques for Parallel Programs, Lecture Notes in Computer Science 191, Springer-Verlag, NY, 1985.

[2] LAMPORT, L., *The 'Hoare Logic' of Concurrent Programs, Acta Informatica*, vol 14, pp 21-37, 1980.

[3] LEVIN, G.M., Proof Rules for Communicating Sequential Processes, PhD thesis, Department of Computer Science, Cornell University, August 1980.

[4] MONTAQUE, R., *The Proper Treatment of Quantification in Ordinary English*, in (Eds) J. Hintikka, J. Moravcsik and P. Suppes, Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics, pp 221-242, Reidel Dordrecht; reprinted in (Ed) R. H. Thomason, Formal Philosophy, Selected Papers of R. Montaque, pp 247- 270, Yale University Press, 1974.

[5] OWICKI, S.S. & D. GRIES, *Verifying Properties of Parallel Programs: An Axiomatic Approach*, CACM, vol 19, no 5, pp 279-285, May 1976.

TESTING AND DEBUGGING OF CONCURRENT SOFTWARE BY DETERMINISTIC EXECUTION

Kuo-Chung Tai and Richard H. Carver
Department of Computer Science
Box 8206
North Carolina State University
Raleigh, NC 27695-8206, USA

ABSTRACT

An execution of a concurrent (centralized, parallel, or distributed) program nondeterministically exercises a sequence of synchronization events, called a synchronization sequence. This nondeterministic execution behavior creates the following problems during the testing and debugging phases of a concurrent program P: (1) When testing P with input X, a single execution is insufficient to determine the correctness of P with input X, and (2) when debugging an erroneous execution of P with input X, there is no guarantee that this execution will be repeated by executing P with input X.

In this paper, we show how to test and debug a concurrent program by forcing deterministic executions of this program according to a given synchronization sequence, referred to as **deterministic execution testing and debugging,** respectively. We first present a language-based approach to deterministic execution debugging. We then discuss the advantages and problems of deterministic execution testing, and describe how to solve these problems by using our language-based approach.

BIOGRAPHICAL SKETCH

K. C. Tai received his Ph.D. degree in Computer Science from Cornell University in 1977. He is currently a Professor of Computer Science at North Carolina State University. He has published papers in the areas of software validation, programming languages, and compiler construction. His current research interests include software testing and the analysis, testing, and debugging of concurrent programs.

Richard Carver recently completed a Ph.D. in computer engineering from North Carolina State University. Dr. Carver received an M.S. in computer studies from North Carolina State University, and a B.S. in computer science from the Ohio State University. His major research interests are in the area of software validation, especially for concurrent software. He is currently involved in the implementation of testing and debugging tools for concurrent Ada programs.

TESTING AND DEBUGGING OF CONCURRENT SOFTWARE BY DETERMINISTIC EXECUTION

Kuo-Chung Tai and Richard H. Carver
Department of Computer Science
Box 8206
North Carolina State University
Raleigh, NC 27695-8206, USA

1. Introduction

The software life cycle of a program includes testing and debugging phases. Testing is the process of executing the program in order to detect errors. Debugging is the process of locating and correcting errors. The conventional approach to testing a program P is to execute P with each selected test input once and then compare the test results with the expected results. If a test input detects the existence of an error, P is usually executed again with the same input in order to collect debugging information. After the error has been located and corrected, P is executed again with each of the previously tested inputs in order to verify that the error has been corrected and that in doing so no new errors have been introduced. (Such testing, called regression testing, is also needed after P has been modified for enhancement.) These testing and debugging approaches are commonly applied to sequential programs; however, they encounter problems when they are applied to concurrent programs.

A concurrent (centralized, parallel, or distributed) program is written in either a concurrent language or a sequential language with a set of concurrent constructs supported by an operating system. Let P be a concurrent program. An execution of P non-deterministically exercises a sequence of synchronization events called a synchronization sequence (or SYN-sequence). This nondeterministic execution behavior creates the following problems during the testing and debugging phases of P:

> Problem 1. When testing a concurrent program P with input X, a single execution is insufficient to determine the correctness of P with input X.

The reason for this problem is that a single execution of P with input X exercises only one of possibly many SYN-sequences of P with input X. Thus, a single execution of P with input X might not detect all of the errors for P with input X. Even if P with input X has been executed successfully many times, it is possible that a future execution of P with input X will produce an incorrect result. Also, after P has been modified in order to correct the error detected by an execution of P with input X, a single successful execution of P with input X implies neither that the detected error has been corrected nor that the correction has introduced no new errors.

During the debugging phase of P, since multiple executions of P with input X may exercise different SYN-sequences, the following problem is encountered:

> Problem 2. When debugging an erroneous execution of a concurrent program P with input X, there is no guarantee that this execution will be repeated by executing P with input X.

In the remainder of this paper, we describe how to test and debug a concurrent program by forcing deterministic executions of this program according to given synchronization sequences. In section 2, we discuss basic approaches to defining SYN-sequences of concurrent programs. In section 3, we give definitions of correctness and errors for concurrent programs. In section 4, we present a language-based approach to deterministic execution debugging. In sections 5 and 6, we discuss the advantages and problems of deterministic execution testing, and describe how to solve these problems using a language-based approach. Section 7 concludes this paper. This paper includes major revisions of our earlier work reported in [Tai85,Car86].

2. Approaches to Defining SYN-sequences of Concurrent Programs

The problem of how to define the format of a SYN-sequence of a concurrent program is referred to as the SYN-sequence definition problem. In this section, we discuss two approaches to defining SYN-sequences of concurrent programs.

Let P be a concurrent program. We define the synchronization objects and synchronization events in P as follows.

Case (a): Assume that processes in P communicate with each other via shared variables. Define a synchronization object in P as a shared object that is used to synchronize processes in P. For example, if P uses a semaphore (monitor) for synchronization, then this semaphore (monitor) is a synchronization object in P. If mutually exclusive accesses of a shared variable in P is not guaranteed, then the shared variable itself is a synchronization object. Define a synchronization event in P as an operation involving a synchronization object. For example, an operation on a semaphore or an entry into a monitor is a synchronization event.

Case (b): Assume that processes in P communicate with each other by sending and receiving messages. The destination (source) designator in a send (receive) operation defines a "communication channel" [And83]. Synchronization is provided by the use of blocking or nonblocking send and receive operations and the use of selective communication statements such as guarded commands or select statements. The communication channels in P are considered as synchronization objects. Define a synchronization event in P as an operation on a channel. For example, a mailbox is a synchronization object and a send or receive operation is a synchronization event. In general, synchronization events may be send and receive commands, higher level message-passing constructs such as remote procedure calls (DP), input/output commands (CSP) and rendezvous (Ada), or "channel status operations" such as checking a channel for an available message or checking to see if a message that was sent has been received.

Assume that P consists of concurrent process P1, P2, ..., Pm and assume that P has synchronization objects O1, O2, ..., and On with two possible synchronization operations OP_A and OP_B. The two basic approaches to defining a SYN-sequence of P are:

Object-based: Let a SYN-sequence of a synchronization object be a sequence of synchronization events involving this object. A SYN-sequence of P is (S1,S2,...,Sn), where Si, 0<i<=n, denotes a SYN-sequence of synchronization object Oi. Such a SYN-sequence of object Oi will (likely) be a sequence of (process identifier,event identifier) pairs; for example, ((P1,OP_A),(P2,OP_B),...) indicates that the first synchronization operation performed on Oi is OP_A by process P1, the second is OP_B by process P2, etc.

Process-based: Let a SYN-sequence of a process be a sequence of synchronization events involving this process. A SYN-sequence of P is (S1,S2,...,Sm), where Si, 0<i<=m, denotes a SYN-sequence of process Pi. Such a SYN-sequence of process Pi will (likely) be a sequence of (object identifier,event identifier,event number) triples; for example, ((O1,OP_A,1st),(O1,OP_A,3rd),...) indicates that process Pi performs the 1st and 3rd operations on object O1 and both are OP_A operations.

In both of these approaches, the SYN-sequence definition of P is based on a partial-ordering since there are separate SYN-sequences for each object/process. These definitions can easily be extended into total-ordering based definitions. The definition of SYN-sequence for a concurrent language L can be language-based, implementation-based, or a combination of both. A language-based definition defines the synchronization objects and events in terms of the language constructs available in L. An implementation-based definition defines the synchronization objects and events in terms of L's underlying implementation. Finally, a SYN-sequence of P may have different formats for different purposes. Generally we define two types of SYN-sequences for a concurrent language or construct. The first type provides complete synchronization information about an execution of a concurrent program. The second type is a simplified form of the first type and is useful for debugging (see later discussion). Below, we show these two types of SYN-sequences for the monitor construct.

For the sake of simplicity, we consider a SYN-sequence as a total-ordering based sequence of

synchronization events. Let P be a concurrent program that consists of processes and monitors. (It is assumed that every shared variable is inside a monitor, that processes delayed on a condition variable are to be waken up in first-in-first-out order, and that mutual exclusion is not released when a nested monitor call is made.) The types of synchronization events that occur during an execution of P include

(a) the start of execution of a monitor procedure by a process,
(b) the completion of execution of a monitor procedure by a process,
(c) the execution of a wait(condition) operation in a monitor procedure by a process, and
(d) the execution of a signal(condition) operation in a monitor procedure by a process.

Thus, an execution of P can be characterized by a sequence of such synchronization events, called a monitor-sequence (or M-sequence). A textual description of an M-sequence is

((H1,P1,D1,N1),(H2,P2,D2,N2),...) (1)

where (Hi,Pi,Di,Ni) denotes the ith, i>0, synchronization event in the sequence, with Hi being the type of this synchronization event, Pi the calling process, Di the called monitor procedure, and Ni the condition variable. This definition is a total-ordering based definition using the object-based approach (with each monitor as an object). It is easy to show that the result of an execution of a monitor-based program P with input X can be determined by P, X, and the M-sequence of this execution.

Define the simple monitor sequence (or SM-sequence) of an execution of a monitor-based program as the sequence of process names corresponding to the sequence in which processes enter monitors during this execution. Thus, a textual description of an SM-sequence is

(P1,P2,...) (2)

where Pi, i>0, is the name of the process making the ith monitor entry. Although the format of an SM-sequence is much simpler than that of an M-sequence, the result of an execution of a monitor-based program P with input X can be determined by P, X, and the SM-sequence of this execution. The reason is that during an execution of P with input X, the sequence of synchronization events inside a monitor M can be determined by the implementation of M, the sequence in which processes in P enter M (via monitor calls), and the values of parameters of these monitor calls.

3. Definitions of Correctness and Errors for Concurrent Programs

In this section we first provide a formal definition of correctness for concurrent programs. Based on this definition, we then define two types of errors in concurrent programs.

3.1 A Definition of Correctness for Concurrent Programs

Let P be a concurrent program. Due to nondeterministic execution behavior, the implementation (or code) of P may allow the existence of two or more distinct feasible SYN-sequences for P with the same input. Let

FEASIBLE(P,X) = the set of feasible SYN-sequences of P with input X.

Since nondeterministic execution behavior is expected, the specification of P may allow the existence of two or more distinct SYN-sequences to be exercised by (the implementation of) P with the same input. A SYN-sequence is said to be valid for P with input X if this SYN-sequence is expected to be exercised by some execution of P with input X, according to the specification of P. Let

VALID(P,X) = the set of valid SYN-sequences of P with input X.

A concurrent program P is said to be correct for input X (with respect to the specification of P) if and only if

(a) FEASIBLE(P,X) = VALID(P,X) and
(b) every feasible SYN-sequence of P with input X produces a correct result.

P is said to be correct (with respect to the specification of P) if and only if P is correct for every possible input. Although the FEASIBLE and VALID sets are, in general, impossible to determine, the above definition of correctness is useful for classifying types of errors in concurrent programs and for studying other issues on the validation of concurrent programs.

3.2 Types of Errors in Concurrent Programs

Based on the above definition of correctness, a concurrent program P is incorrect for input X if and only if one or more of the following conditions hold:

(a) FEASIBLE(P,X) is not equal to VALID(P,X). Thus, either
  (a.1) there exists at least one SYN-sequence that is feasible, but invalid for P with input X, or
  (a.2) there exists at least one SYN-sequence that is valid, but infeasible for P with input X. (Or both (a.1) and (a.2) are true.)

(b) There exists an execution of P with input X that exercises a valid (and feasible) SYN-sequence, but produces an incorrect result.

An error in P that causes condition (a) to be true is said to be a synchronization error. (The term "synchronization error" and its synonyms, such as "timing error" and "time-dependent error", have been used in previous literature on testing and debugging concurrent programs without having formal definitions.) An error that causes condition (b) to be true is said to be a computation error.

After the completion of an execution of a concurrent program P with input X, we need to determine the existence of synchronization and computation errors. Assume that this execution exercises a (feasible) SYN-sequence, say S, and produces a result, say Y. Then one of the following conditions holds:

(a) S is valid and Y is correct,
(b) S is invalid and Y is correct,
(c) S is valid and Y is incorrect,
(d) S is invalid and Y is incorrect.

Note that condition (b) implies the production of a correct result from an incorrect SYN-sequence. This "coincidental correctness" may be rare, but it is not impossible. Conditions (b) and (d) imply the existence of a feasible, but invalid SYN-sequence (and thus a synchronization error). And condition (c) implies the existence of a computation error. (Condition (d) may be caused by a combination of synchronization and computation errors.) Thus, during the testing of a concurrent program P, we should collect the SYN-sequences of executions of P and then determine the validity of these SYN-sequences. We will show later that the collected SYN-sequences of P can be used for other testing and debugging activities.

The problem of determining whether a SYN-sequence is valid for a concurrent program with a given input is referred to as the SYN-sequence validity problem. Solving this problem requires a comparison between a given SYN-sequence and the specification of P. The validity of a SYN-sequence can be determined automatically only if the specification is written formally.

4. APPROACHES TO DETERMINISTIC EXECUTION DEBUGGING

The purpose of deterministic execution debugging is to allow an erroneous execution to be replayed so that debugging information can be collected. Such information can be collected as it is needed by using a traditional interactive debugger in conjunction with the replay facility.

Deterministic execution debugging involves the following three problems:

(a) SYN-sequence definition: How to define the format of a SYN-sequence of a concurrent program so that the SYN-sequence provides sufficient information for repeating this execution. (Here, a SYN-sequence does not include complete synchronization information because such information is more than sufficient for repeating this execution.)

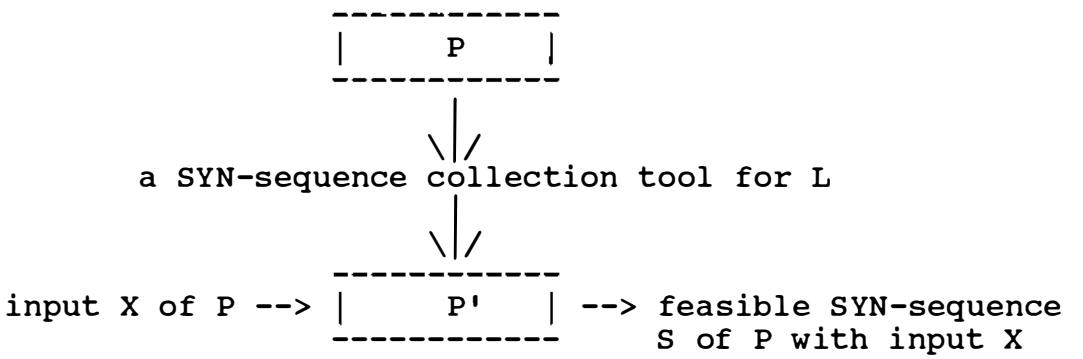
(b) SYN-sequence collection: How to collect the SYN-sequence of an execution of a concurrent program?

(c) SYN-sequence replay: How to repeat the SYN-sequence of a previous execution of a concurrent program? (This problem was referred to as the reproducible testing problem[1] [Bri78,Tai85].) Let the SYN-sequence exercised during an execution of a concurrent program with a given input be called a feasible SYN-sequence of this program with this input. Thus, the SYN-sequence replay problem deals with how to repeat a feasible SYN-sequence of a concurrent program.

Below we first briefly describe a commonly used approach to deterministic execution debugging and then present a language-based approach.

Implementation-Based Approach. The implementation of a concurrent language L usually consists of three components: the compiler, the run-time system, and the operating system. An implementation-based approach is to modify some or all of the three implementation components of language L so that during an execution of a program P (written in L) with input X, the sequence of synchronization events can be controlled according to a given feasible SYN-sequence of P with input X. For example, in [LeB87] the SYN-sequence replay problem for parallel programs using read and write operations on shared objects was solved by modifying the implementation of read and write commands. Also, implementation-based solutions are used in a number of debuggers that have been developed for various concurrent languages and operating systems that support concurrency ([Bri89],[GarH84],[GarM85],[Gil88],[Mau85],[McD88],[Pro88],[Wit88]). They allow the programmer to directly control execution by performing "scheduler-control" operations such as setting breakpoints, selecting the next running process, rearranging processes in various queues, and so on. Since implementation-based tools are dependent on a particular compiler, run-time system or operating system, it may be difficult if not impossible to port such tools from one implementation to another. Also, due to the complexity of the implementation components and the level of detail, it may be difficult to express the basic design of an implementation-based tool in such a way as to be useful for developing this tool on possibly radically different implementations. Thus, an implementation-based approach may force tool developers to reinvent the wheel for each new implementation.

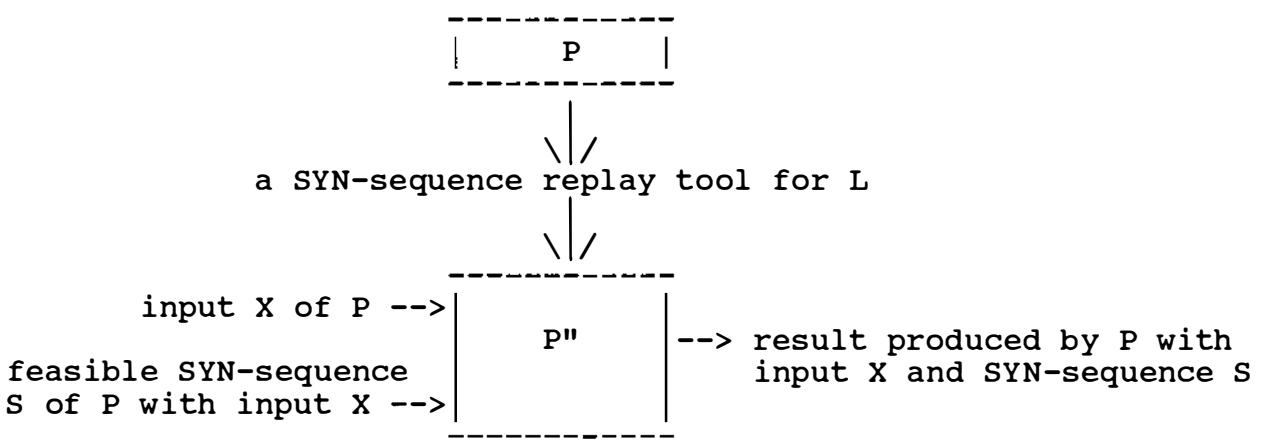
Language-Based Approach. Let L be a concurrent language. First, we define the format of a SYN-sequence of a program written in L in terms of the synchronization constructs available in L so that a SYN-sequence provides sufficient information for deterministic execution debugging. The definition of an SM-sequence for the monitor construct in section 1 is an example. Second, we develop a SYN-sequence collection tool for L, which is illustrated by the following diagram:



The SYN-sequence collection tool for L transforms a program P written in L into a slightly different program P' (also written in L) so that P' is equivalent to P except that during an execution of P', the SYN-sequence of this execution is also collected. Third, we develop a SYN-sequence replay tool for L, which is illustrated by the following diagram:

---

[1] The term "reproducible testing" was first used in [Bri78] and later in [Tai85,Car86,Tai86b,87a]. This term, however, often causes confusion since it is more a debugging issue than a testing issue.



The SYN-sequence replay tool for L transforms P written in L into a slightly different program P" (also written in L) so that any execution of P" with (X,S) as input, where S is a feasible SYN-sequence of P with input X, definitely exercises S and produces the same result as P with input X and SYN-sequence S would.

A language-based approach to deterministic execution debugging has the following advantages:

(a) The transformation of a concurrent program for SYN-sequence collection or replay produces a program written in the same language. Thus, this approach does not create a portability problem.

(b) The definition of a SYN-sequence and the development of a SYN-sequence collection or replay tool for a concurrent language are independent of the implementation of this language.

(c) The source transformation for SYN-sequence collection or replay for a concurrent language L can serve as a high-level design for L's implementation-based debugging tools. (Implementation-based solutions have the advantage of having less run-time overhead because the control of synchronization events is performed at the implementation level instead of the source level.) Also, a language-based solution to SYN-sequence collection or replay may suggest an efficient implementation-based solution.

The concept of source transformation for collecting traces of a concurrent program has been used in a number of debuggers [Ger85,Hel85,McD88]. However, the source transformation for SYN-sequence replay is more difficult than that for trace collection due to two reasons. One reason is that the format of a SYN-sequence must be carefully defined to provide the necessary and sufficient information for SYN-sequence replay. The other reason is that the replay of a SYN-sequence requires the control of execution of synchronization events and thus is more difficult than the collection of a SYN-sequence. To avoid any possible confusion, we point out that the work by German, Helmbold, and Luckham did not deal with the replay problem and their definition of a trace does not provide sufficient information for replay. We have studied the SYN-sequence definition, collection, and replay problems for several concurrent constructs and languages, including Ada [Tai86,89,Car89], Concurrent C [Pat88], send/receive [Tai87], and semaphores and monitors [Car86].

5. Approaches to Testing Concurrent Programs

Testing refers to the execution of P for detecting errors, verifying corrections of errors, and regression testing. In this section, we first describe a commonly used approach to testing concurrent programs, and then present a new approach.

The multiple execution testing approach is the following:

(1) Select a set of inputs of P,

(2) For each selected input X, execute P with X many times and examine the result of each execution.

Multiple, nondeterministic executions of P with the same input may exercise different feasible SYN-sequences and thus are likely to detect more errors. One method for increasing the chance of

exercising different SYN-sequences of P with the same input is to insert delay statements into P with the length of each delay randomly assigned during multiple executions of P. (Delay statements may be supported by a concurrent language or an operating system.)

The deterministic execution testing approach is defined as follows:
(1) Select a set of tests, each of the form (X,S), where X and S are an input and a SYN-sequence of P respectively.
(2) For each selected test (X,S),
  (2.1) determine whether or not S is feasible for P with input X by attempting to force a deterministic execution of P with input X according to S, and
  (2.2) if S is feasible, examine the result of this execution.
Note that in this testing approach, a test for P is not just an input of P; it consists of an input and a SYN-sequence of P and is referred to as an IN-SYN test. The problems of how to select SYN-sequences and how to determine the feasibility of SYN-sequences are discussed later.

The deterministic execution testing approach provides the following advantages over multiple execution testing:

(a) This approach allows the use of carefully selected SYN-sequences to test a concurrent program P and can detect the existence of valid, but infeasible as well as invalid, but feasible SYN-sequences of P. Multiple execution testing of P can only detect the existence of invalid, but feasible SYN-sequences of P; and it might not exercise some of the feasible SYN-sequences of P that would be selected for deterministic execution testing.

(b) Assume that an error is detected by an execution of P. After an attempt has been made to correct the error, P can be tested with the input and SYN-sequence of this erroneous execution in order to make sure that the error has been corrected.

(c) After P has been modified for correction or enhancement, P can be tested with the inputs and SYN-sequences of previous executions of P in order to make sure that the modification does not introduce new errors.

Although deterministic execution testing (DET) has advantages over multiple execution testing (MET), it requires additional effort for the selection of SYN-sequences and the determination of feasibility of SYN-sequences. The amount of such effort can be reduced by combining DET and MET. Below are several possible strategies for combining DET and MET to test a concurrent program P:

(a) Apply DET in the module testing of P and MET in the system testing of P.

(b) In the module or system testing of P, apply MET first until the test coverage of the module or program reaches a certain value. Then apply DET to reach a higher test coverage.

(c) In the system testing of P, apply DET to certain portions of this program and MET to the remainder of the program.

(d) Apply MET for detecting errors and DET for verifying that errors have been corrected and for regression testing.

The use of MET (DET) for the selection of SYN-sequences is similar to the use of random testing (special value testing) for the selection of inputs. Theoretical and experimental studies of random testing (of sequential programs) show that random testing should be supplemented with the use of carefully selected inputs [Dur84,Vou84a,b]. This suggests that MET should be supplemented with the use of carefully selected inputs and SYN-sequences in order to achieve high reliability of concurrent software. To illustrate the need for combining MET and DET, consider the final concurrent Ada program in [Hel85] for solving the gas station problem. This program was executed many times without showing a deadlock. (An execution resulting in a deadlock indicates the existence of a feasible, but invalid SYN-sequence.) But a deadlock was found later by applying deterministic execution testing [Tai86a].

6. Problems in Deterministic Execution Testing of Concurrent Programs

Deterministic execution testing involves several problems not encountered in multiple execution testing.

Two of these problems, SYN-sequence definition and collection, were addressed earlier in the discussion of deterministic execution debugging. In this section we briefly discuss two other problems.

SYN-sequence Selection Problem

The SYN-sequence selection problem is how to select SYN-sequences that are effective for error detection. To test a concurrent program P with input X, we may select SYN-sequences of P as follows:

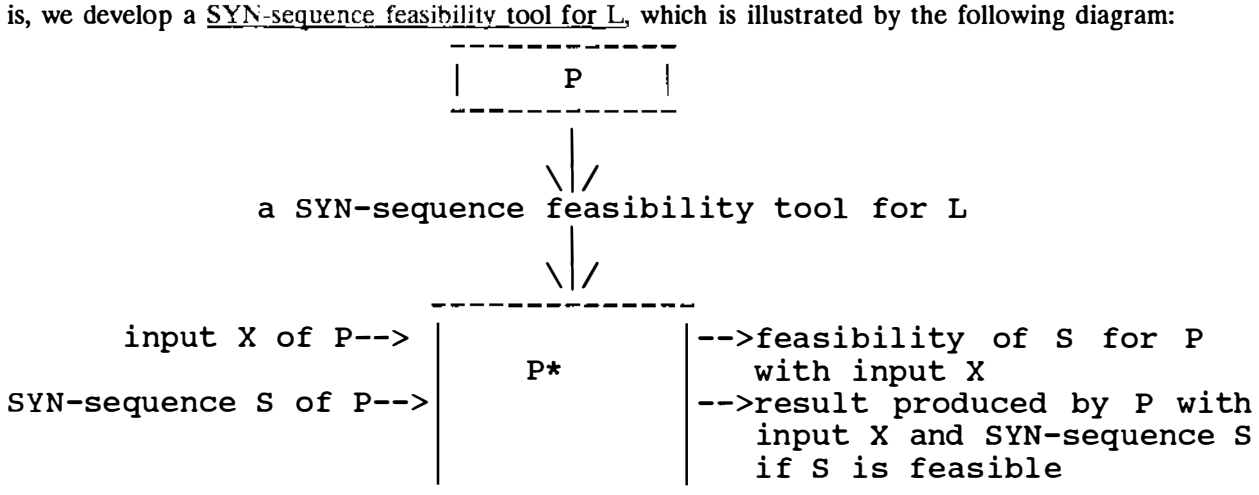
(a) According to the specification of P, select a number of SYN-sequences that are valid for P with input X. We also select a number of SYN-sequences that are invalid for P with input X. Several methods for generating test sequences of communication protocols based on the finite state machine model have been developed [Sar84]; they only use test sequences that are guaranteed to be repeatable without having to force a deterministic execution. If the set of synchronization events in the specification of P is different from that in the implementation of P, then a specification-based SYN-sequence of P needs to be transformed into a program-based SYN-sequence of P.

(b) According to the implementation of P, select a number of SYN-sequences of P with input X. (For each of these SYN-sequences, we need to determine whether it is valid for P with input X.) In [Tay86] a number of path selection criteria were defined for concurrent programs. (A path of a concurrent program defines a SYN-sequence of this program.)

More research is needed for specification-based and program-based selection of SYN-sequences. Due to the complexity of SYN-sequences of a concurrent program, it is important to develop techniques and tools to reduce the effort for selecting SYN-sequences that are effective for error detection.

SYN-Sequence Feasibility Problem

For each selected SYN-sequence of P with input X, we need to determine whether it is feasible for P with input X. The problem of how to determine whether a SYN-sequence is feasible for a concurrent program with a given input is referred to as the SYN-sequence feasibility problem. To solve this problem, we can apply the language-based approach presented in section 4 for SYN-sequence collection and replay. That is, we develop a SYN-sequence feasibility tool for L, which is illustrated by the following diagram:



The SYN-sequence feasibility tool for L transforms P written in L into a slightly different program P* (also written in L) so that an execution of P* with (X,S) as input, where X and S are an input and SYN-sequence of P respectively, determines whether or not S is feasible for P with input X and produces the same result as P with input X and SYN-sequence S would, provided that S is feasible. The transformation of P for SYN-sequence feasibility is similar to, but slightly complicated than that for SYN-sequence replay for two reasons. One reason is that the former is based on a type of SYN-sequence of L that provides complete synchronization information. The other reason is that the former needs to check the feasibility of each given SYN-sequence.

Now we describe how to use P* to solve the SYN-sequence feasibility problem for P. Let S be a selected SYN-sequence of P with input X. Assume that S satisfies one of the following three conditions: (If S does

not, extend S or select a prefix of S to produce such a SYN-sequence.)

(1) S is expected to be exercised completely and after that, no more synchronization events are expected and P is expected to have a normal termination. In this case, add "#NT" to the end of S.

(2) S is expected to be exercised completely and after that, no more synchronization events are expected and P is expected to have an abnormal termination. In this case, add "#AT" to the end of S. (An abnormal termination can be caused by various types of errors such as divide-by-zero, deadlock, expiration of allocated CPU-time, etc.) If the cause of abnormal termination needs to be identified, use a distinct symbol for each possible cause of abnormal termination.)

(3) S is expected to be exercised accordingly except that the last synchronization event in S is expected to be impossible to occur. In this case, add "#IN" to the end of S. This case is useful for SYN-sequences that are expected to be infeasible.

Let S' be the SYN-sequence collected during an execution of P* with (X,S) as input. (The termination symbol at the end of S is ignored during the execution of P*.) S' is modified as follows:

(a) delete synchronization events in S' exercised by statements in P* that are not in P.

(b) The termination condition of this execution has three possibilities:

  (b.1) normal termination. In this case, add "NT" to the end of S'.

  (b.2) abnormal termination caused by statements in P* that are also in P. In this case, add "AT" to the end of S'.

  (b.3) abnormal termination caused by statements that are in P* but not in P. This abnormal termination is a deadlock due to wait for an event in S to occur. In this case, add this event and "#IN" to the end of S.

Then S is feasible if and only if S = S' and S (or S') ends with "#NT" or "#AT". (For the sake of simplicity, earlier definitions of SYN-sequences did not include an indication of the termination condition.) It is possible to define other forms of SYN-sequences in order to specify more precisely "what is expected" and "what has happened". By doing so, it would be easier to identify any difference between "what is expected" and "what has happened".

Note that our method for determining the feasibility of a SYN-sequence is not completely correct since the SYN-sequence feasibility problem is, in general, undecidable. (The SYN-sequence feasibility problem is undecidable because it is impossible to determine whether a program has an infinite loop.) In our method we assume that when an execution of a program reaches the end of the allowed CPU-time, the operating system will abort this execution and report this error as an expiration of CPU-time. Thus, our method fails to determine the feasibility of a SYN-sequence S of P with input X only if an execution of P* with input (X,S) does not have an infinite loop and the allocated CPU-time for P* is not long enough for the completion of this execution. (An alternative to the detection of expiration of CPU-time is that P* issues a "time-out" message when the interval between the most recent synchronization event and the current time is longer than the allowed interval between two consecutive synchronization events.)

Earlier we defined M-sequences and SM-sequences of monitor-based concurrent programs and we saw that SM-sequence provide sufficient information for SYN-sequence replay. One important question is "For monitor-based programs, do we develop the SYN-sequence feasibility tool to determine the feasibility of M-sequences, SM-sequences, or both?" Our answer is that the tool should be developed to determine the feasibility of M-sequences, not SM-sequences. As mentioned earlier, an M-sequence provides complete synchronization information, but an SM-sequence does not. As a result, the feasibility of an M-sequence S is not necessarily the same as the feasibility of the SM-sequence S' that is extracted from S (for the same program with the same input). If S is feasible, then S' is also feasible, but not vice versa.

7. Conclusion

As concurrent software becomes more widespread, the problem of how to effectively test and debug concurrent software becomes more critical and thus has recently received much attention. In this paper we have discussed the need for deterministic execution testing and debugging of concurrent programs and presented a langauge-based approach to deterministic execution debugging and deterministic execution

testing. Deterministic execution testing and debugging require effort for solving some problems not encountered in conventional testing and debugging approaches, but such effort is needed in order to achieve higher reliability and lower cost of concurrent software. (About half of the development cost of a software system is expended in testing.)

Fig. 1 shows the use of the SYN-sequence replay, collection, and feasibility tools to create an environment that supports multiple execution testing, and deterministic execution testing and debugging of a concurrent program. Note that it is possible to combine these three tools into a single one that transforms program P into a program that can serve as P', P" or P* depending on the programmer's choice. Based on our language-based approach, we have implemented SYN-sequence collection, replay and feasibility tools for semaphores, monitors, send/receive, Ada, and Concurrent C. Some of these tools have been used in a graduate level course on software engineering for concurrent systems, which is taught by the first author of this paper at North Carolina State University [Tai88]. Students in the course expressed that these tools were very helpful for validating concurrent programs.

As mentioned earlier, language-based solutions to the problems in deterministic execution testing and debugging can serve as a high-level design for implementation-based solutions. Presently, concurrent programs are written by using (1) a concurrent language, (2) a combination of a sequential language and the concurrent constructs supported by an operating system, or (3) a combination of a concurrent language and the concurrent constructs supported by an operating system. (Case (3) exists when a concurrent language does not provide sufficient concurrent constructs for a specific application.) In case (2) or (3), we can consider the combination of the (sequential or concurrent) language and the operating system constructs as a concurrent language, develop language-based solutions and then transform these solutions into implementation-based solutions.

Finally, we would like to mention that deterministic execution testing and debugging can be applied to validate real-time programs that involve concurrency. The correctness of a real-time program has two aspects: logical correctness and timing correctness [Wir77,Gom86]. Deterministic execution testing and debugging can certainly be used to validate the logical aspect of a real-time program. However, the impact of run-time overhead on the timing characteristics of a real-time program needs to be considered when validating the timing correctness of this program.

Acknowledgement
The authors wish to thank Dr. Evelyn Obaid for comments that helped improve the clarity of this paper.

References

[Bri78] Brinch Hansen, P., "Reproducible testing of monitors," Software-Practice and Experience, Vol.8, 1978, 721-729.

[Bri79] Brindle, A. F., Taylor, R. N., and Martin, D. F., "A Debugger for Ada Tasking," IEEE Trans. Soft. Eng., Vol. 15, No. 3, 1989, 293-304.

[Car86] Carver, R. H., and Tai, K. C., "Reproducible testing of concurrent programs based on shared variables," Proc. 6th Int. Conf. on Distributed Computing Systems, 1986, 428-433.

[Car89] Carver, R. H., "Testing, debugging, and analysis of concurrent software," Ph.D. thesis, North Carolina State University, 1989.

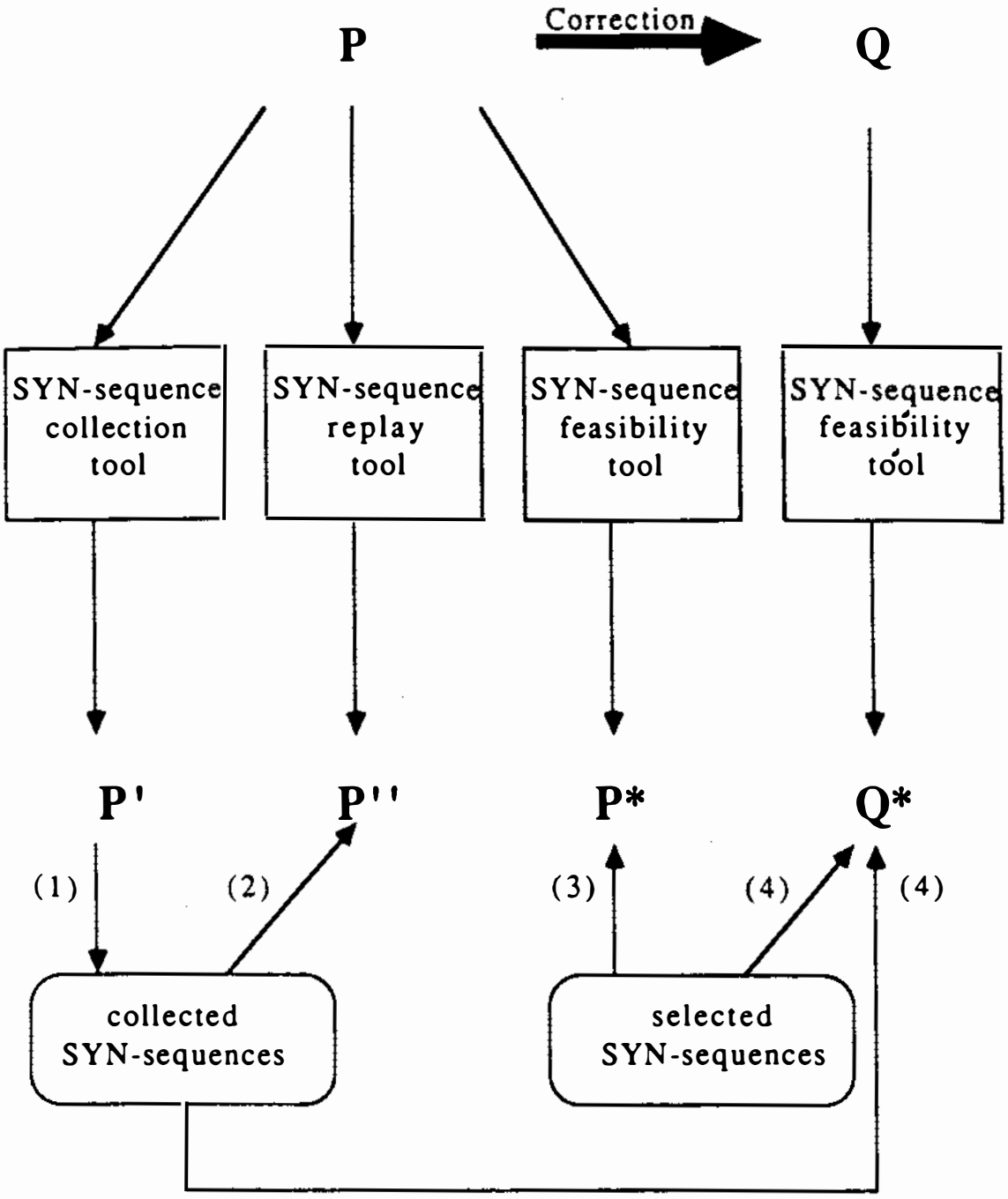
[Dur84] Duran, J. W., and S. Ntafos, "An evaluation of random testing," IEEE Trans. Soft. Eng., Vol. SE-10, July 1984, 438-444.

[GarH84] Garcia-Molina, H., et al., "Debugging a distributed computing program," IEEE Trans. Soft. Eng., Vol. SE-10, No. 2, March 1984, 210-219.

[GarM85] Garcia, M. E., and Berman, W. J., "An approach to concurrent systems debugging," Proc. IEEE 5th Inter. Conf. on Distributed Computing Systems, May 1985, 507-514.

[Ger84] German, S. M., "Monitoring for deadlock and blocking in Ada tasking," IEEE Trans. Soft. Eng., Vol. SE-10, No. 6, Nov. 1984, 764-777.

[Gom86] Gomaa, H., "Software development of real-time systems," Communications ACM, July, 1986.
[Gil88] Gilles, J., and Ford, R., "A Window Based Debugger for a Real Time Ada Tasking Environment," Proc. 5th Washington Ada Symposium, June 1988, 59-67.
[Hel85] Helmbold, D., and Luckham, D., "Debugging Ada tasking programs," IEEE Software, Vol. 2, No. 2, March 1985, 47-57.
[LeB87] LeBlanc, T. J., and Mellor-Crummey, J. M., "Debugging parallel programs with instant replay", IEEE Trans. Computers, Vol. C-36, No. 4, April 1987, 471-482.
[Mau85] Mauger, C., and Pammett, K., "An event-driven debugger for Ada," Proc. Ada Inter. Conf. (ACM Ada LETTERS, Vol. V, Issue 2, Sept./Oct. 1985), 124-135.
[Pat88] Patwardhan, M. R., and Tai, K.C., "A Debugging Environment for Concurrent C," Technical Report TR-88-12, Dept. of Computer Science, North Carolina State University, 1988.
[Sar84] Sarikaya, B., and Bochmann, G. v., "Synchronization and specification issues in protocol testing", IEEE Trans. on Communications, Vol. COM-32, No. 4, April 1984, 389-395.
[Tai85] Tai, K. C., "On testing concurrent programs," Proc. COMPSAC 85, Oct. 1985, 310-317.
[Tai86a] Tai, K. C., "A graphical representation of rendezvous sequences of concurrent Ada programs," ACM Ada Letters, Vol. VI, No. 1, Jan. & Feb. 1986, 94-103.
[Tai86b] Tai, K. C., and Obaid, E. E., "Reproducible testing of Ada tasking programs" Proc. IEEE 2nd Int. Conf. on Ada Applications and Environments, April 1986, 69-79.
[Tai87] Tai, K. C., and Ahuja, S., "Reproducible testing of communication software", Proc. IEEE COMPSAC '87, Oct. 1987, 331-337.
[Tai88] Tai, K. C., "A course on software engineering for concurrent systems", Proc. Second Software Engineering Institute Conference on Software Engineering Education, Lecture Notes in Computer Science, Vol. 327, Springer-Verlag, 1988, 106-119.
[Tai89] Tai, K. C., Carver, R. H., and Obaid, E., "Deterministic execution debugging of concurrent Ada programs," to be published in Proc. COMPSAC 89, September 1989.
[Tay86] Taylor, R. N., and Kelly, C. D., "Structural testing of concurrent programs," Proc. Workshop on Software Testing, July 1986, 164-169.
[Vou86a] Vouk, M. A., McAllister, D. F., and Tai, K. C., "An experimental evaluation of the effectiveness of random testing of fault-tolerant software," Proc. Software Testing Workshop, July 1986, 74-81.
[Vou86b] Vouk, M. A., Helsabeck, M. L., McAllister, D. F, and Tai, K. C., "On testing of functionally equivalent componenets of fault-tolerant software", Proc. COMPSAC '86, Oct. 1986, 414-419.
[Wir77] Wirth, N., "Toward a discipline of real-time programming," Communications ACM 20(8), 1977, 577-583.
[Wit88] Wittie, L., "Debugging Distributed C Programs by Real Time Replay," ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, May 1988, 57-67.



(1) multiple execution testing of P
(2) deterministic execution debugging of P
(3) deterministic execution testing of P
(4) verification of correction to P

Figure 1. Use of SYN-sequence Collection, Replay, and Feasibility Tools

A COMMON BASIS FOR INDUCTIVE INFERENCE AND TESTING

LEONA F. FASS

Abstract

Inductive inference is concerned with determining a correct model or implementation of a specified behavior S, given a suitable behavioral sample. Testing is concerned with determining suitable experiments to show whether an implementation designed to fulfil a specification S is incorrect. We describe circumstances where inference and testing are complementary processes producing "essentially the same" result. Then effective testing is achievable precisely when effective inference is, so an implementation of a specification S can be verified.

The theory we develop to show testing and inference are "essentially the same" confirms results in functional or specification-based software testing. It should have utility in the design of sound approximating testing techniques, and the development of software testing theory.

Biographical Sketch

Leona F. Fass received a B.S. in Mathematics and Science Education from Cornell University, and an M.S.E. and Ph.D. in Computer and Information Science from the University of Pennsylvania. Prior to receiving her Ph.D. she held administrative, research and teaching positions at Penn and Temple University. Since then she has been on the faculties of the University of California, Georgetown University, and the Naval Postgraduate School. Her research interests include application and adaptations of inductive inference results to the practical domain.

The author may be contacted at mailing address:

Leona F. Fass
P.O. Box 2914
Carmel, CA 93921

*This research was partially supported by the NPS Foundation Research Program.

1. Introduction

Inductive inference of correct programs from I/O examples, and testing for incorrectness through I/O experiments, clearly are related processes. A link to verification could be established were it possible to show that a program or system had been tested enough to determine it completely error-free. In that case, demonstrative testing could "prove" its correctness by seeking all, and finding no, errors. Such verification would be achievable if only we knew the testing-experiments that were necessary to identify all "bugs". Then, when we ultimately produced software determined to be error-free by the characterizing tests, we'd be assured it would always behave correctly.

Despite the obvious hypothesized above, in the application area of software development complex programs and systems always do have "bugs", and procedures to detect them are generally unsatisfactory. There are no proven techniques for generating or selecting a data sample or testing experiments that can guarantee identification of all errors. Hence there is no way of determining, should a testing procedure detect no incorrectness, whether the software tested is genuinely error-free or, more likely, whether something is wrong with the tests.

A theory of testing is now under development (e.g., by Weyuker [17]) and it may ultimately establish which, and when, experiments that test for incorrectness indeed have been adequate. Only once such adequacy criteria are determined might a program or system be assessed, convincingly, as correct -- in the sense of having been tested sufficiently. Defining the appropriate data and experiments for program error detection is a long-standing problem acknowledged to be very difficult (most would say, impossible) to solve. Paraphrasing Hamlet [14], we describe it as a problem of characterizing the infinite set of all possible instances of incorrect program behavior by means of a finite set of tests.

Like testing, inductive inference is dependent, for its success, on characterizing infinite possible behavior in a finite fashion. It is concerned with constructing or discovering models of correctness from appropriate data or experiments demonstrating how the models should (or should not) behave. This concept of constructing correctness has been applied in such areas as: machine synthesis; automatic programming; knowledge acquisition; automated reasoning and theorem-proving; and the field of computational learning being formalized today. With decades of study by (among others) machine theorists, logicians, artificial intelligence

researchers and now, complexity experts, there is a wealth of successful inference results based on considerable formal theory.

In contrast to testing, extensive inductive inference research has determined precise characterizing data and experiments sufficient for determining correct models in numerous behavioral domains. Relationships between inference and testing are thus worthy of investigation [4,7,10,12], with the goal of adapting relatively well-developed inference theory to testing's practical domain.

We compare inference and testing processes, primarily motivated by our discussions with Cherniavsky and by his "Popperian" view [4] of software testing procedures. His thesis is that testing software systems and detecting (inevitably present) defects is not to be considered a negative result, i.e. just an instance of finding that "things have gone wrong". Rather, identifying "bugs" is a positive result of testing, since it leads to the development of better, more correct systems.

From a generally recursion-theoretic orientation, and his familiarity with testing technology, Cherniavsky's "Popperian" approach and his work with Smith, Statman and Velauthapillai [4-7] has brought analogues of inductive inference results into the realm of testing. By contrast, our perspective comes from experience within inductive inference research [8,9,11] where, generally, our orientation has been logical and algebraic.

In the problems we consider we find inductive inference and testing to be "essentially the same" processes, with a common basis in realization theory (if a processor, program, etc., deterministically produces or processes exactly a specified behavior, it is a realization of that behavior) [3]. Within this theoretical framework, we compare inference and testing and determine circumstances where inference-related testing indeed does lead to verification, in the sense described above.

By constraining the problem space we find that if a realization of a behavior is inferable from a finite behavioral sample, then a potential realization is effectively-testable for incorrectness. Furthermore the correct behavioral sample sufficient for inference defines the data sample sufficient for testing. Due to the constraints we impose, we may conclude that by performing appropriately on all tests (doing what it should and not doing what it shouldn't) the potential realization of the behavior will

always behave correctly. In that sense it is verified.

It is only because we constrain the problem environment that we can show inductive inference and testing are "essentially the same", and that testing can "verify" by exhaustively demonstrating correctness (or, no incorrectness). It is satisfying to know inferable models are effectively testable within our logical-algebraic framework, for this confirms similar results found from other mathematical perspectives (e.g., [4,5,7]).

But even if the constraints we applied within the theoretical environment could be applied within the practical programming and system development environment, we would not expect this theory to be immediately applicable to practice. After all, knowing that the data in a conclusive test set is dependent on the data needed for inference, means that the sufficient testing of a complex software system is as likely as the possibility we could infer it.

However, even if our theory is not automatically implementable, it should assist in devising theoretically sound approximating testing techniques. This would improve upon ad hoc techniques often used today, and would be consistent with developing testing theory.

2. An Informal View of Inductive Inference and Testing

Inductive inference is informally described as reasoning from specifics to the general. It is the process we use to learn infinite bodies of knowledge from samples or examples of the knowledge to-be-acquired.

For instance, as children we all learn to speak a natural language, by observing others speak. Observing enough, we acquire a basic vocabulary. Then we learn to put words together into phrases and later, grasp the concept of correct sentence structures. We do this even though we generally aren't told we're being given examples of sentences in a natural language or what the language might be. Without any formal education in grammar we ourselves infer, through observations of the examples, some knowledge of syntactic structure of the language to which we are exposed. As long as we acquire vocabulary and structural knowledge, we have the potential of generating or recognizing an infinite set of sentences, almost all of which we've never heard before. We've inferred them, from a finite set of examples.

Another instance of such inference is the process we'd use to learn the infinite sequence of integers represented by the sample 2, 4,..., or by the alternate sample 2, 4, 8, 16,

32, ... If we observed the appropriately chosen (to characterize the sequence) sample, we'd perform (mental) experiments to determine the pattern of the sequence it illustrated. Given the first sample we might conclude the sequence is even integers greater than zero, with the n^{th} element computable from the formula 2n. Given the second, we might conclude the sequence is non-zero integer powers of 2, with the n^{th} element computable from the formula 2^n. Assuming the sequence is the latter, we'd be more likely to infer this from the second sample than from the first. Having done so, we'd have acquired the infinite knowledge in the sequence. That is, we'd be able to produce its n^{th} element for given n, even if we've never seen it in the sequence, or otherwise, before.

Testing may be informally described, within the present context, as finding specifics to disconfirm (or confirm) an hypothesized generalization. We have familiarity with a body of knowledge and are given a possible characterization that we test for consistency with what we know. If a test detects incorrectness, then we disconfirm the given potential characterization. Otherwise, if satisfied we've tested enough, we may accept it as correct.

Suppose, for instance, we "know" the sequence 2, 4, 8, 16, 32, ..., and we are given, as potential characterization of each n^{th} element, the formula 2n. Most likely we'd choose some specific values of n, compute 2n in each case, and compare the results with the corresponding elements in the sequence that we know. If we test with values n = 1 or n = 2, we won't discover that the given formula is incorrect. If we test with n = 3, we detect incorrectness immediately, and might ask for another potential characterization of the sequence to test.

Similarly, suppose we "know" the sequence and are given a program claimed to compute each n^{th} element for input n. Assuming no immediately obvious errors noted by inspection, we might execute the program to test it. We would choose appropriate values of n to input and compare output results with what we know they <u>should</u> be. If we found a mismatch between the program's output for specific input n and our expectation, we'd go back to look at the program and try to correct it. If we chose tests appropriately and the output always confirmed our expectations, we might be satisfied that the program is correct.

In the inference instances described above, we are "the inferrer" acquiring infinite knowledge from the finite sample or examples an informant (teacher, parent, textbook author, etc.) provides. We are dependent on the informant, who has

the knowledge, to present suitably representative data, that will lead us to correctly generalize. In the testing instances we are "the tester", and we have acquired the knowledge to be characterized. We are responsible for determining a set of suitably representative data, to see whether or not an hypothesized generalization is incorrect. With a "Popperian approach" [4] we expect incorrectness, and our goal is to demonstrate the hypothesis is wrong. But, once satisfied we cannot demonstrate incorrectness, we conclude that the hypothesized generalization may be correct.

3. A Formalization of Inference and Testing Processes

Inductive inference has been studied in computer science, from various mathematical perspectives, and from an artificial intelligence orientation -- as a basis for machine learning. Inference-and-testing relationships have recently been investigated mathematically, as in Cherniavsky, Statman and Velauthapillai's approach, describing each process in terms of two-player games [7]. Our own approach to inductive inference, adapting concepts of algebra and logic, has been constructive: we have investigated problems where a correct characterization of knowledge can be "built" from the structure represented in an appropriate sample. We extend this approach to testing and then, to inference-and-testing relationships, formalizing the notions described above.

Terminology:

We describe the body of knowledge to be acquired as a behavior. Any correct characterization of a behavior (a description, a process or processor producing it) is model. For a given behavior, we describe a process or processor exactly producing it (generally deterministically) as a realization. A sample consisting of elements of the (correct) behavior is positive data. Examples of incorrectness, relative to the behavior and the domain in which it lies, is negative data. With these concepts we reformulate the inference and testing processes informally illustrated in Section 2.

An inductive inference process is concerned with discovering a model M of a specified behavior S, given only a sample or examples. In constructive approaches a characterizing sample of S, i.e., positive data, is observed and generalized to find an M producing or realizing exactly S, the entire behavior. The technique is particularly suitable for modeling infinite behavior with an underlying structure,

demonstrable in a finite sample (and where S is finitely-realizable). Experiments defined by the sample identify the structural components M must possess to produce exactly S. We would not expect a unique result from the process, since behaviorally equivalent M and N need not be structurally equivalent. But constraints on the process may ensure that a unique or "best" M realizing S (e.g., a minimal realization) is found.

In the problems we consider, the goal of inductive inference thus is to determine the correct or "best possible" structure of a "black box" M with behavior S, when only a sample of S has been observed.

Instances of the process include:

- o inferring a grammar for a regular or context-free language L, from a suitably chosen language sample;
- o inferring a recognitive processor, e.g., the lexical analyzer component of a compiler, from sample strings it is to analyze;
- o inferring the structure of a minimal sequential machine with output, from examples of its I/O behavior;
- o inferring a "simplest" program P to compute a function f on integers x, from selected x, f(x) pairs.

Example 1

In a typical programming language, a simple identifier might be informally described as "a letter, followed by any number of letters or digits". Even if "letter" is restricted to just a and b, and "digit" to just 2, the resultant specified set S of identifiers,

S = {a, aa,..., a2,..., b,...ab,..., bba,..., bba2a,...}

is infinite.

But the informal description of S provides enough information about its underlying (syntactic) structure to define a finite grammar producing exactly the elements of the infinite set S, and nothing else. One such grammar is G whose constructs are: syntactic categories <id>, <letter>, <digit>; and production rules

<id> → <letter> | <id><letter> | <id><digit>
<letter> → a | b
<digit> → 2

Furthermore, S's structure, as conveyed by its description of the above grammar, implies it is processible by a finite recognitive device, such as the lexical analysis component of a compiler. Such a device R may be designed with recognitive-state and transition constructs to deterministically accept exactly S, and nothing outside of S such as 22 or 2a. An R so designed realizes S as its behavior.

The constructs of G or R are determined by the strucure of S, as provided in its description. Each construct corresponds to some feature of S, as it's been specified. In an inductive inference process no information about S would be given, other than a finite selection of its strings, e.g., a, b, a2, b22a. An appropriately chosen sample must represent the features of S's syntactic structure sufficiently so that constructs necessary in a model may be inferred. Choosing the "right" sample of S is essential to construction of "black box" behaviorally-equivalent to the finite R or G described above. Since we know there are finite constructs in a model for S, S's syntactic features can be represented in a finite sample. We will describe below a sample of S sufficient for inference of models behaviorally-equivalent to R or G.

A testing process experiments with a given candidate model, M', of behavior S to see if it fulfils that specification. Functional or "black box" testing externally observes how M' behaves in experiments (e.g., its output on selected inputs), while structural or "white box" testing considers the components of M' (e.g., whether all its paths or branches are exercised) as well as its externally observed experimental behavior. The testing process has a model of the specified behavior S, so it knows how M' should always behave. It must choose appropriate experiments sampling S to discover whether, or where, M' may be incorrect.

The implied goal of testing thus is to detect incorrectness of a given "black box" or "white box" M' designed to fulfil specification S, so that a more correct model of the behavior may be found. However, should no experiment detect incorrectness and if experimentation is deemed sufficient, correctness of M' might then be verified [1, 4, 7, 10].

Example 2

A testing process might be concerned with detecting incorrectness of a grammar G', or recognitive device R', claimed to characterize the set S of simple identifiers described in Example 1. The process would have some specification of S, such as its informal description, and would seek to determine whether it was fulfilled by G' or R'. It would test to see if G' generates, and R' accepts, only elements of the specified S, such as a, b, aa, a2, b22a. It would also test to see that G' does not generate, nor does R' accept, anything outside of S, such as 22 or 2ab.

If the testing process observes that G' can't generate, or R' won't accept, some element it knows to be in S such as a2, it detects incorrectness. If it observes G' generate, or R' accept, something it knows is not in S, such as 22, it detects incorrectness. But as long as no test reveals incorrectness of G' or R', the process must determine whether to continue testing the "black" or "white box" G' or R', or to cease and consider it to be correct.

Just as the set S of simple identifiers is infinite, so is the set of strings not in S (e.g., it contains all strings of 2's). A satisfactory testing process must choose an appropriately characterizing finite sample from each set to test G' or R' effectively. If this can be done, the process may conclude that if incorrectness is not detected on the finite tests, G' or M' will never behave incorrectly and so must be correct.

A constructive inductive inference process, as in the "black box" paradigm we have described, seeks an M correctly realizing or producing (infinite) specified behavior S, given an appropriate (finite) behavioral sample (positive data). On the other hand, a testing process seeks to determine, through a (finite) set of suitable tests, if an M' claimed to produce or realize S is incorrect. At first glance inference and testing seem complementary notions. But common sense reasoning tells us it is far easier to characterize correctness in a finite sample or model, than to categorize and finitely characterize all the possible ways things can go wrong [13].

The software testing literature (e.g., as reviewed in [1, 4, 7, 14, 16, 17]) presents numerous examples of "white box" or program-based testing techniques, "black box" or

specification-based techniques, and combinations thereof, intended to discover inevitable software "bugs". No technique is claimed to detect all incorrectness, a goal which, if achievable, could lead to development of verifiably (by default) correct software.

However, corresponding to some of the "black box" constructive inference examples described above are examples of effective "black box" testing. By suitably constraining the problem space, it can be shown that for some classes of specified behaviors S: if a correct M producing S is inductively inferable from finite data, then a candidate M' claimed to produce S is effectively testable for incorrectness. In such cases, inference and testing indeed are complementary; verification of a candidate M' claimed to have correct behavior S follows directly from finite testing that demonstrates M' is not incorrect. We discuss such situations next.

4. When Inference and Testing are "Essentially the Same"

Here we describe circumstances where effective testing is achievable precisely when effective inference is. Then for a specified behavior S, both inference and testing will lead to discovery of a correct M producing exactly that behavior. Inference will lead to discovering a correct M by constructing it from a finite behavioral sample. Testing will discover a correct M by sufficiently testing a candidate using a finite set of characterizing data. Since inference and testing then achieve the same outcome -- discovering correctness -- and do so in a finite fashion, under these circumstances we may conclude that inference and testing are "essentially the same" process.

To illustrate how effective testing and effective inference may correspond, we will outline our approach to a specific problem where we found this to be the case (and describe those conditions under which specific results may be generalized). We established constraints on the problem space to develop a "black box" constructive inference technique: building a unique finite realization of an infinite behavior from the information contained in a finite behavioral sample. We then discovered that the constraints making constructive inference of a unique "black box" possible automatically led to a technique for effective "black box" testing. We found that by developing a successful inference paradigm we'd actually developed an inference/testing paradigm, where the data sufficient for inference prescribes the data sufficient for effective, conclusive tests.

Our inference/testing paradigm came out of our work in language inference. In [8, 9, 11] we investigated linguistic inference problems, to determine syntactic models for languages within certain classes. Our original goal was to find characterizing grammars or recognitive devices, given only samples of the languages to be characterized (e.g., a set of sentences or statements). Our particular interest was in the regular and context-free syntactic classes which include much of programming language in use today, e.g., Pascal. So while we first investigated this inference problem as a language acquisition problem within the scope of artificial intelligence, results in this area are applicable to (automated) compiler design [12].

A regular, or a context-free, language is a (generally) infinite set of sentences, statements or syntactic structures over a finite vocabulary. It lies within an infinite domain of structures over the same vocabulary, some of which are in the language, the rest of which are not. We let S denote all "correct structures", i.e., structures in the language to be characterized, and D the domain of all structures in which S lies. If we seek a recognitive device (we restrict ourselves to recognition here) M characterizing S, we wish it to accept all of S and nothing else, i.e., nothing in D - S. If we want to infer M, we must be able to construct its components given only the information found in a finite data sample. We need to construct a finite M from an appropriate selection of data, chosen from within an infinite domain.

To make this problem solvable and "narrow down" the space of possible solutions, we originally showed that with constraints, a unique M could be found. We established that for any S within domain D, a unique minimal realization of S, $\hat{M}$, could be defined. Thus what we sought to infer was the structure of a unique, smallest possible deterministic "black box" $\hat{M}$ that would accept all of S and accept nothing else in D.

We first showed that the components (states) of $\hat{M}$ corresponded to classes of the elements of D, and that they could be determined by distinguishing experiments involving positive (S) and negative (D - S) data. We then showed that the components $\hat{M}$ needs to process acceptable behavior (S) could be found from only positive data, with the remaining structure of the "black box", to process and "trap" unacceptable behavior (D - S), determinable by default.

We ultimately showed that <u>if</u> S has a finite realization with n components, then $\hat{M}$ must have $\leq$ n components (since it is minimal), determined by distinguishing experiments involving positive data (elements of S) of "length $\leq$ 2n".

This is the finite sample of the behavior S sufficient to construct the "black box" $\hat{M}$, accepting exactly S and nothing else (in D - S). Components of $\hat{M}$ to process negative data (in D - S) are still determinable, as noted above, by default.

Since we can easily define the complement of the characterizing positive data within domain D, effective testing follows directly from effective constructive inference. Suppose we are given an n-state candidate minimal realization of S, M', and we want to test to see if it behaves incorrectly. To behave correctly, the "box" M' must accept all of S and nothing in D - S. Now, we know correct behavior of an n-state processor is characterized by experiments involving all "length $\leq$ 2n" elements of S. Since we've delimited our domain we can show incorrect behavior is characterizable by the complement set (of negative data): all "length $\leq$ 2n" elements of D - S.

Hence, we can show that if M' is tested on all "length $\leq$ 2n" elements of D, it will be sufficiently tested. If M' accepts all "length $\leq$ 2n" elements of S, it produces correct behavior. If M' accepts no "length $\leq$ 2n" element of D - S, it produces no incorrect behavior. It can be shown that if M' performs in this fashion on the specified finite set of tests, it will <u>always</u> behave correctly. So M' will thus be verified.

We note these results directly adapt to inference of a "minimal" n-variable grammar $\hat{G}$ generating S, and to testing a candidate n-variable "minimal" grammar, G'. We may infer $\hat{G}$ from length $\leq$ 2n elements of S . We may test G' by seeing it generates all $\leq$ 2n elements of S and seeing it generates no $\leq$ 2n elements in D - S. Then G' will have behaved correctly on these finite experiments and <u>it</u> will be verified.

Example 3

The set S of simple identifiers described in Examples 1 and 2 above is properly contained in the infinite domain D consisting of "all strings of any number of a's, b's and 2's". Within this domain, we may describe S as "any string beginning with an a or b" and D - S as "all other strings in D", i.e., those that don't begin with a or b.

From the G or R described in the examples above, it can be seen that S is a context-free language and, in fact, is regular. It can be shown that S has a 3-state minimal realization $\hat{M}$, accepting all of S and nothing in D - S. One state, q_0, is the start state; one state q_1, is a final, accepting state that $\hat{M}$ enters, and remains in, if an input starts with a

or b; the last state q_2 is a "dead state" that $\hat{M}$ enters and remains in if an input starts with neither a nor b.

It can be seen that each string in D when input to $\hat{M}$ has a specific state of $\hat{M}$ as its response. Thus the states are actually representatives of disjoint (equivalence) classes of strings in D: q_0 represents the empty string; q_1 represents the strings in specified set S, such as b, bbab2; q_2 represents all else in D, such as 2, 2ab2. State-to-state transitions may be seen as representatives of string concatenations: the empty string (represented by q_0) suffixed with 2 yields 2 (represented by q_2, which is "equivalent" to a transition in $\hat{M}$ from q_0 to q_2 on input 2; 2 (represented by q_2) suffixed with ab2 yields 2ab2 (represented by q_2), "equivalent" to a q_2-to-q_2 transition on input ab2; bb (represented by q_1 suffixed with ab2 yields bbab2 (represented by q_1), "equivalent" to a q_1-to-q_1 transition on input ab2.

These equivalences between strings of D and the constructs of the minimal realization for S provide the key to inference. To find the necessary components of a minimal realization for S, we need only find the equivalence classes of strings these components represent. (It can be shown strings x and y are equivalent, and representable by the same state, if they can prefix the same strings z and produce resultant strings in S. That is, xz is in S exactly if yz is. Thus a and bb could be in the same class, but 2 and bb could not since only one of 2ab2, bbab2 is in S. ab2 "distinguishes" 2 from bb in a concatenation experiment).

Since we know $\hat{M}$ has 3 states, these correspond to 3 equivalence class of strings, and it can be shown that representatives sufficient to determine these classes are found in the length ≤ 3 elements of D. State-to-state transitions, determine from string-to-string concatenations, is described above, these are discoverable from concatenating the length ≤ 3 strings to each other. So we need only examine length ≤ 6 strings to discover the components of $\hat{M}$ by inference. However all strings in D that are not components of strings in S fall into the same string class under the equivalence described above (since they aren't components of S, no z suf- onto them will ever put them in S). We need not examine them in our experiments to construct $\hat{M}$: they automatically fall into the "dead state" class in $\hat{M}$, which is determinable by default. However, this negative data is precisely what we need for testing. It's what a 3-state realization for S must not accept.

To infer $\hat{M}$, a minimal realization for the specified behavior S, the set of simple identifiers, we need only examine length ≤ 6 strings of S (positive data) which will be more than sufficient to find $\hat{M}$'s components other than those involving "the dead state". The latter is found by default.

To test a candidate 3-state M' to see if it realizes S, we must see if it accepts the positive data that determines a correct realization: all strings in S of length ≤ 6. Then we must see that it does not accept any negative data that a 3-state realization must not accept. This is all of D - S of length ≤ 6.

We note that corresponding to $\hat{M}$ is a minimal $\hat{G}$ producing exactly the set S of simple identifiers. It has one syntactic category, $\langle id_1 \rangle$ and productions:

$$\langle id_1 \rangle \longrightarrow a \mid b \mid \langle id_1 \rangle\, a \mid \langle id_1 \rangle\, b \mid \langle id_1 \rangle 2$$

It is inferable from all strings in S of length ≤ 2. Then any other 1-variable grammar claimed to produce S is testable against all strings of D of length ≤ 2. It should generate those in S, and none in D - S.

In relation to the original problem we sought to solve, these results mean that for any language in the class we considered, we can inductively infer a unique, minimal syntactic characterization or effectively test a candidate characterization for incorrectness. In relation to inference and testing in general, these results are based on constraints that enable us to conclude:

> If we can infer a realization of an infinite specified behavior S from a finite (positive data) behavioral sample, then we can effectively test a candidate realization of S for incorrectness.

5. Application to Practical Program Testing

Our inference/testing paradigm is successful because in the cases we consider:

- o domains are delimitable (so we know what is and isn't correct behavior);
- o domains are denumerable (so we can generate precise requisite samples);
- o membership and other necessary queries are decidable (so we can perform experiments, and see whether results are in S or in D - S);
- o behaviors can be defined functionally (so we have no problems of non-determinism and can apply algebraic theory to construct realizations);

- o behaviors are finitely-realizable (so they, and their complements, can be characterized by finite samples).

Similar and complementary results on general classes of testable and inferable sets and functions are described from a different theoretical perspective in [4, 7].

In practice, the constraints on the theoretical problem space described above, that make inference and testing "essentially the same", do not apply in actual programming or systems development environments. Programs are not simply semantics-free implementations of functional behaviors. Nor can we enumerate, answer decision queries about, or even delimit the domains in which their behaviors lie. The constraints or conditions that make our inference/testing paradigm successful simply aren't present, nor can they be expected to be present, in actual applications.

If they were, we'd be able to show that:

> If a simplest program $\hat{P}$ realizing a specified behavior S (or, fulfilling the specification) is inferable from a finite sample of S, then a candidate program P' claimed to realize S (or, fulfil that specification) is effectively testable from a finite sample of S's containing domain.

The logician would question why we'd even bother to test P', when we might as well infer a correct and simple $\hat{P}$ fulfilling S in the first place. The complexity expert would point out that the intractability of the inference and testing processes for any complicated program make either process infeasible. The inductive inference expert would note the extremely limited success to date in program inference, making a testing process that's dependent on complementary inference not likely to be generally applicable.

But some of the problems inherent in the inference and testing paradigm just noted are problems inherent in the testing process alone, and this is one of the reasons determining sufficient testing is so difficult. Since no testing strategy has been found to guarantee detection of all errors, all techniques in use, if not ad hoc, are at best approximations to conclusive testing. Our theory should assist in extending such approximative testing techniques.

6. Concluding Remarks

We are satisfied with our inference/testing theory since it confirms results in functional or specification-based testing described in [5-7], found via a different theoretical approach. Thus we feel our results should have utility in the development of additional approximating testing techniques, based on what we see as sound underlying theory.

For example, just as distinguishing experiments on behavior samples find components of a realization of a correct specified behavior, similar experiments could roughly classify incorrectness. This would be consistent with the relatively successful approximating testing techniques discussed in [15], based on testing from categories of errors.

The concept of inference-adequate testing developed by Weyuker [16] is even more consistent with our inference/testing paradigm, and might be expanded upon for developing testing techniques and theory. She describes an iterative process of inferring programs from test data, comparing the results with an actual program to-be-tested, and with the specification it was designed to fulfil. She thus shows that program inference may be used to approximate the equivalence of a program and its specification, a problem that is generally undecidable. In that case, as in ours, inductive inference and testing are "essentially the same".

Acknowledgements

We thank Elaine Weyuker and John Cherniavsky for introducing us to the area of testing. Without discussions with John, we would not have investigated the inference/testing relationships described above. We also appreciate Dick Hamlet's encouraging remarks and suggestions for improvements to an earlier draft of this paper.

References

1. Adrion, W.R., M.A. Branstad and J.C. Cherniavsky, "Validation, Verification and Testing of Computer Software", Computing Surveys 14 (1982), 159-192.

2. Angluin, D. and C.H. Smith, "Inductive Inference Theory and Methods", Computing Surveys 15 (1983), 237-269.

3. Bobrow, L.S. and M.A. Arbib, Discrete Mathematics: Applied Algebra for Computer and Information Science, W.B. Saunders, Philadelphia, 1974.

4. Cherniavsky, J.C., "Computer Systems as Scientific Theories: A Popperian Approach to Testing", Proc. of the Fifth Pacific Northwest Software Quality Conference, Portland, October 1987.

5. Cherniavsky, J.C. and C.H. Smith, "Using Telltales in Developing Program Test Sets", Georgetown University Department of Computer Science Series, TR-4, 1986.

6. Cherniavsky, J.C. and C.H. Smith, "A Recursion Theoretic Approach to Program Testing". IEEE Transactions on Software Engineering, 13 (1987), 777-784.

7. Cherniavsky, J.C., R. Statman and M. Velauthapillai, "Testing and Inductive Inference: Abstract Approaches", Georgeown University Department of Computer Science Series, TR-5 1987. Also appears in Proceedings of the First Workshop on Computational Learning Theory, Morgan-Kaufmann, 1988.

8. Fass, L.F., "Learning Context-Free Languages From Their Structured Sentences", SIGACT News 15 (1983) 24-35.

9. Fass, L.F., "Inference of Skeletal Automata", Georgetown University Department of Computer Science Series, TR-2, 1984.

10. Fass, L.F., "Remarks on Inductive Inference and Testing", presented at the Assocation for Symbolic Logic 1988-89 Annual Meeting, University of California, Los Angeles, January 1989.

11. Fass, L.F., "Learnability of CFLs: Infering Syntactic Models from Constituent Structure", presented at the 1987 Linguistics Institute Meeting on the Theoretical Interactions of Linguistics and Logic, Stanford, July 1987. Research note appears in SIGART Special Issue on Knowledge Acquisition, April 1989, 175-176.

12. Fass, L.F., "On Language Inference, Testing and Parsing", presented at 1989 Linguistics Institute, Meeting on the Theoretical Interactions of Linguistics and Logic, University of Arizona, Tucson, July 1989.

13. Fetzer, J.H., "Program Verification: The Very Idea", Communications of the ACM, 31 (1988), 1048-1063.

14. Hamlet, R., "Special Section on Software Testing", Communications of the ACM, 31 (1988), 662-667.

15. Ostrand, T.J. and M.J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests", Communications of the ACM, 31 (1988), 676-686.

16. Weyuker, E.J., "Assessing Test Data Adequacy through Program Inference"; ACM Transactions on Programming Languages and Systems, 5 (1983), 641-655.

17. Weyuker, E.J., "The Evaluation of Program-Based Software Test Data Adequacy Criteria", Communications of the ACM, 31 (1988), 668-675.

AUTOmate™: A New Approach to Software Verification

Northon Rodrigues, William Tracy, Jr. & Gregory Miller

...!sun!nosun!cvedc

Prime
Computervision Division
Electronics Development Center
Beaverton, Oregon

Abstract

This paper describes the design and implementation of the AUTOmate™ system for testing highly interactive integrated CAE software. As software products in electronic design automation grow increasingly complex, test and verification of product stability and quality becomes correspondingly complex and time-consuming. Assuring quality, completeness, and timeliness of testing becomes particularly difficult when dealing with highly interactive and integrated products. Automated software testing tools, therefore, are becoming essential. The need for automated software testing tools led the authors to develop a general-purpose "software shakedown tool" that could facilitate the testing of any software product produced by Prime Computer.

Biographies

Northon Rodrigues received his Bachelors of Science in Computer Science from Portland State University, Portland, OR. in 1986. He is presently a software engineer for Prime Computer in the Electronics Development Center. Mr. Rodrigues is also pursuing graduate work in computer science, and has published training materials on the UNIX operating system, and as of August 1989, on C++. He currently oversees the testing and quality assurance efforts for the EDA (Electronics Design Automation) products developed in Beaverton, Oregon.

William Tracy, Jr. received his Bachelors of Science in Mathematics from Portland State University, Portland, OR. in 1987. He is presently a Software Engineer for Prime Computer in the Electronics Development Center. His primary responsibility is the C-Graphics Interface, which ensures graphics applications portability. Previously, Mr. Tracy was in the Quality Assurance and Product Testing Group.

Gregory Miller received his Bachelors of Science in Computer Science from the University of Oregon, Eugene, OR. in 1982. He has completed graduate coursework in human interface design, including summer seminars at Stanford University in '85 and '86. Mr. Miller also recently received his law degree (J.D.) emphasizing technology law, in May 1989. He has worked as a software engineer and technical writer for nearly 8 years, and currently is the senior technical writer for Prime Computer in the Electronics Development Center. He is responsible for tutorial and training materials for Prime's EDA products, and in addition, participates in the design and development of human interface subsystems.

Introduction

Prime Computer offers a comprehensive set of electronics design automation tools for printed circuit board design and layout. Since customers demand these products to be seamlessly integrated into a comprehensive design tool environment, our testing strategies must address the testing of the tools in the same manner as used by our customers; that is, while performing individual program testing, we must provide product integration testing.

Today, it is incumbent upon a quality testing program to provide not only discrete testing, but also to provide for testing the integration of products within their typical operating domains. As discussed below, we have had to rethink our testing strategies. As a result, no longer can tools be tested in a vacuum, for it is well settled that the point of tool interface is always the most probable location for error. Further complications arise when these same tools use human interface components such as the mouse and menu "point and click" paradigm. Our previous testing technologies, while allowing for capture and playback of program interaction, still failed to adequately test production code as used by a customer. The methods did not address an integrated tool environment, produced huge log files, and lacked flexibility.

The use of command scripts as an alternate testing mechanism has its own problems. We have no way for verifying the interface between the different applications such as our analog/digital waveform editor and our schematics editor. Moreover, none of the interactivity of the tools can be tested when the command scripts are used, since they do not address mouse clicks and movements. Another problem we face is the lack of control of the tool once the script is started, since the application does not return control to the user (or tester) until the command script has ended.

AUTOmate ™ [2] is designed to address these issues. It completely simulates the actions of a user at the controls of any EDA tool produced by Prime Computer, by replaying mouse and keyboard interaction of the appropriate tool at the appropriate time. This tool is truly "program independent," thereby virtually eliminating the need to "hard wire" the testing module into the product code. The result is that we test *real* production code, *not* test code. A test engineer need only create a test suite in a *record–mode*, and add additional details with a text editor to verify the software.

Moreover, AUTOmate manages the required interprocess communication (IPC) and timing necessary to test Prime's EDA tools. In the remainder of this paper, we review these problems and solutions in more detail, and then present the AUTOmate solution.

2. AUTOmate is a trademark of Prime Computer, Inc..

The Principal Problem

In the software marketplace, time-to-market is a critical ingredient to the success or failure of a product. Severe time constraints and tight development schedules place a strain on the test engineer in the process of software quality assurance. As a result, automated software verification is an essential element of a software quality assurance program.

The basic process of automatic software validation involves running the software with predefined test cases and comparing the output to a set of "known good" results produced by careful data analysis of the output of a previous run of the software.

To fully and accurately test a program, the various internal parts and external commands need to be exercised in the same ways as the program would be used. Providing data files and keyboard input to a program is straight forward, but realistic, reproducible ways of providing mouse input is not as well defined. What is needed is a way to emulate mouse and keyboard input to the program in such a way that the program would "think" that it has received those inputs from an actual user.

The Unified Input Solution

Perhaps the most flexible way to collect mouse/keyboard input is to provide a single *unified input queue*. This single queue can contain mouse and keyboard inputs as well as tool to tool or interprocess communications data. The location of a keystroke verses a mouse action can be preserved according to the *position* in the queue. The data in this *unified input queue* can be stored away for future use. By restoring (or reloading) the contents into a *system input queue*, the data is again available to applications. The key advantage here is the applications need no special code to use these inputs, because they always rely on data from the system input queue (refer to Figure 1.).

Another equally important aspect to this approach is how the events (mouse and keyboard input) are placed in the *unified input queue* (UIQ). This approach requires a *display graphics manager* to pass the events to the UIQ as well as to the application. In the case of Prime Computer's EDA tools, the graphics manager is *C-Graphics*, a Prime Computer proprietary applications level graphics description language. Unfortunately, on a hardware platform where the system window manager does not facilitate access to these events at the system level, one is forced to access them at the applications level. Since our standard platform is the Sun Microsystems workstation line, we must access events through our C-Graphics environment [1]. C-Graphics, therefore, contains the necessary mechanism for passing the events to the application, and simultaneously to the UIQ. As indicated in Figure 1, the UIQ contents (events) are available for subsequent reloading into system queues to once again be passed on to their respective applications. From the applications perspective, the events are as good as real mouse and keyboard events, when in fact, they are from a virtual input mechanism, the UIQ.

The UIQ relies on the graphics manager to collect events. Events are handled in precisely the same manner regardless of the application (due to the presence of C-Graphics). Therefore, one testing mechanism can be constructed for all applications running under a particular graphics management scheme.

This is the conceptual leap that led us to the AUTOmate solution for integrated automatic test and verification. As a result, AUTOmate *is* the *unified input queue* solution.

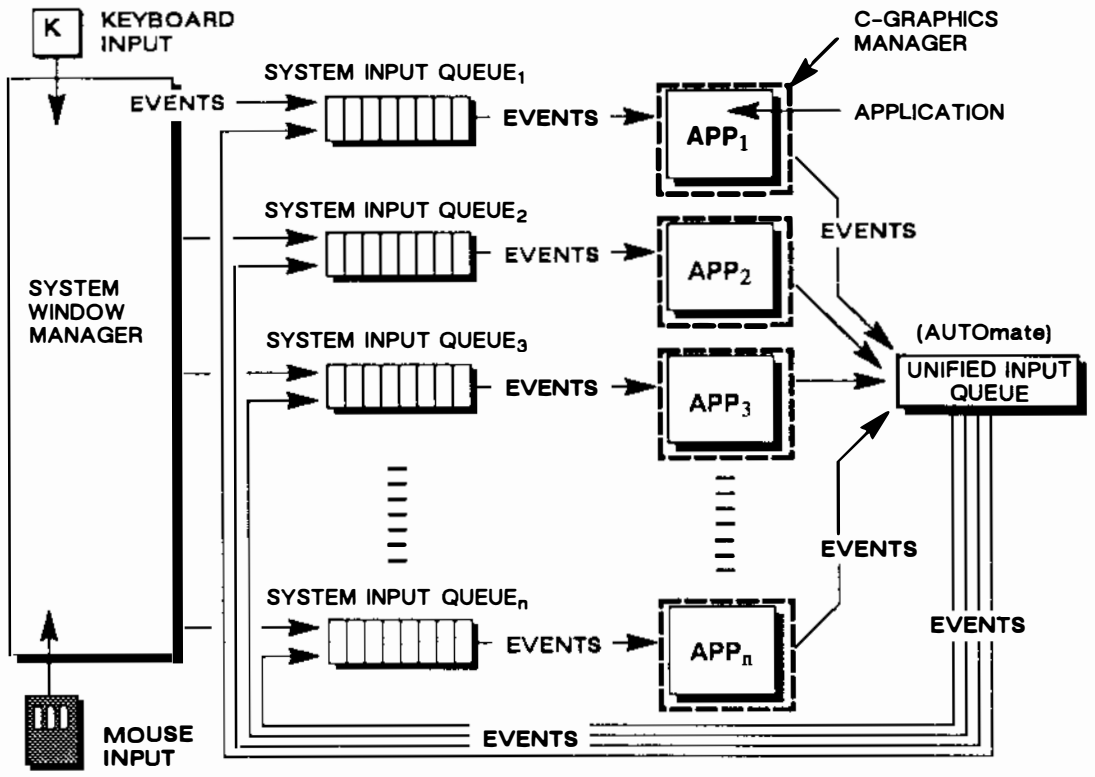


Figure 1. The Unified Input Queue Approach

AUTOmate

In order to accommodate record, playback, and automatic verification functionality across our EDA tool set, it was necessary to design a new tool rather than merely add some special command, or a command line option to our existing EDA tools to collect data.

The result is AUTOmate, a separate program from the applications that manages all interprocess communications, pre-processes and decodes data before sending it to different applications, and encodes incoming data so that it can be easily read and edited by test engineers.

AUTOmate allows the test engineer to construct a test file in a *record-mode* which collects instructions from the desired applications, based on keyboard and mouse input. AUTOmate orders events arriving from different tools in the same sequence as they are used while retaining all inner time dependencies. Only the necessary events are recorded, thereby reducing the size of the test file by eliminating extraneous irrelevant data (e.g., no-ops, null operations, etc.).

A principal design goal was to create a human interface that offered an easy to use, intuitive tool. Using established human interface design principles, we wanted an analogy that would model what we were attempting to accomplish. In essence, what AUTOmate does is record every user interaction

with an application, and then facilitates playback of the interaction to test new versions of an application. What other device in our world of electronics media offers us the ability to record and subsequently play it back? The Video Cassette Recorder (VCR), of course, which become our analogical model for AUTOmate.

The VCR analogy was chosen because of its good fit with the capabilities of the desired tool. The challenge for the human interface designers was to abide by the analogy as much as possible in the development of the tool for testing of software. Early beta testing suggests we have been successful, and the resulting human interface is shown in Figure 2, below.

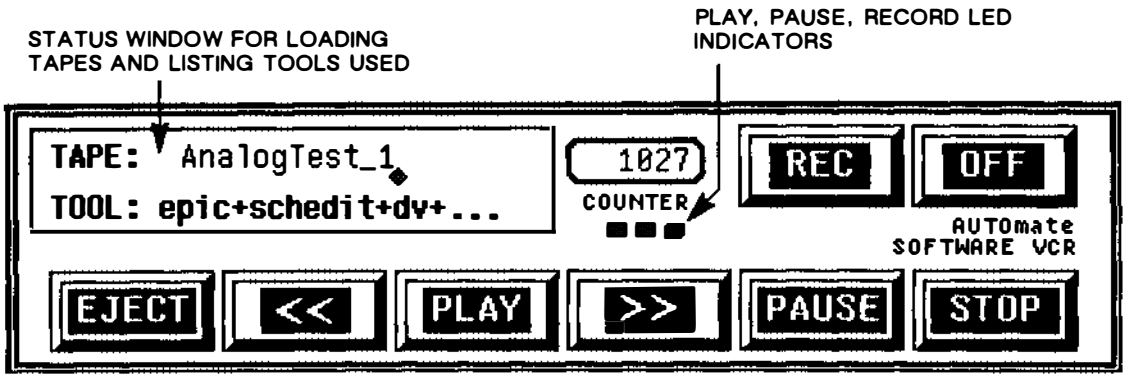


Figure 2. AUTOmate Human Interface

AUTOmate's Human Interface

The STATUS Window. In every step of design and implementation, our objective has been to create a tool the test engineer can use in the same way as a VCR is used to record or play a video program. Therefore, like the slot where one inserts a tape into a VCR, the STATUS Window indicates which tape file is loaded, and what format the tape file is currently using. Here is an example of a number of instances where we were able to successfully enlarge the scope of the analogy without destroying its value. In the VCR world, there are two primary formats, Beta and VHS. Those formats define the nature of the data on a VCR tape.

Likewise, in essence, AUTOmate supports a number of formats (because a command sent to our schematic editor might make no sense if sent to our waveform editor). Therefore, the TOOL STATUS LINE describes the format of data currently being played or recorded. The TAPE STATUS LINE indicates the name of the tape file loaded.

The Counter. AUTOmate employs a counter as does a VCR. The primary purpose of the counter is to assist with the maintenance of the log file (also referred to as a test file). Test files often contain comments, external utility calls, systems calls, etc. The counter provides a precise line number of an instruction or event in the file.

The RECORD Button. Clicking on this button toggles the RECORD mode on or off. A menu pops up when the record button is selected and held displaying the EDA applications that can be recorded. When AUTOmate is in RECORD mode, a red LED is displayed below the Counter.

The OFF and EJECT Buttons. The EJECT button removes the tape file from AUTOmate, clears the name in the STATUS Window, and verifies whether to write-protect the tape. The OFF button exits AUTOmate.

The REWIND Button. The REWIND button "rewinds" the tape file to the previous frame. A frame is a defined point in the log file containing all the information necessary to start playing back the tools from that point on. When the right mouse button is clicked and held, a pop-up menu displays the method of rewinding.

The PLAY Button. The PLAY button plays the loaded file. When AUTOmate is in a PLAY mode, a green LED is displayed below the counter.

The FAST FORWARD Button. This button "advances" the log file. When this button is clicked and held, a pop-up menu displays the method of advancing. The events in the log file can be speeded up, slowed down, or skipped.

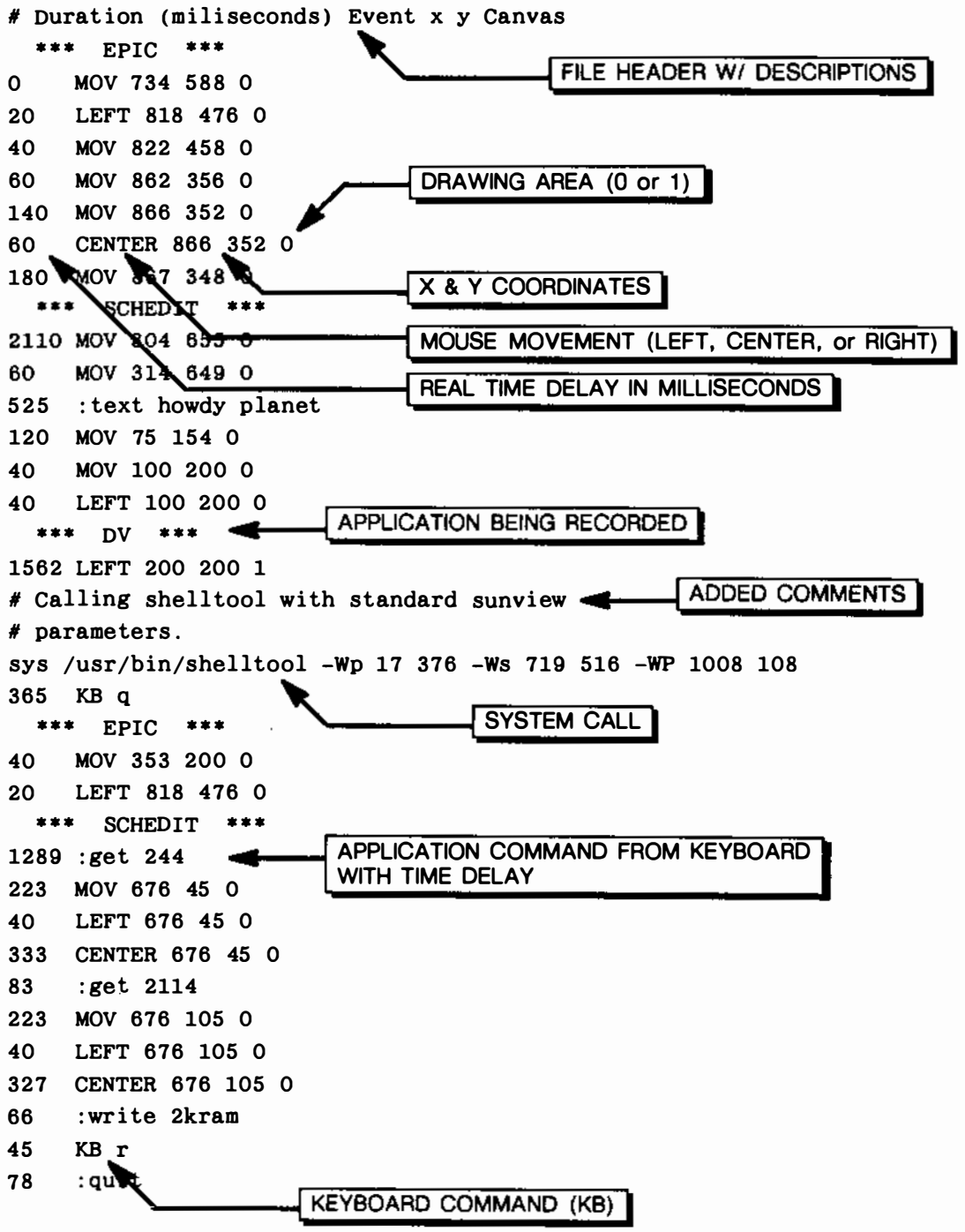
The PAUSE and STOP Buttons. The PAUSE button pauses AUTOmate either in a RECORD or PLAY mode. A yellow LED is displayed below the counter to indicate this state. The STOP button stops AUTOmate in whatever mode it is in.

AUTOmate's Log File

Events sent to the applications under the C-Graphics manager are in turn sent to AUTOmate and recorded in the log file. These same events are reloaded into the system queue when AUTOmate is in playback mode.

The sample log file below shows how the unified input queue (figure 1) is implemented.

Sample AUTOmate Log File




```
# Duration (miliseconds) Event x y Canvas
  ***  EPIC  ***
0    MOV 734 588 0
20   LEFT 818 476 0
40   MOV 822 458 0
60   MOV 862 356 0
140  MOV 866 352 0
60   CENTER 866 352 0
180  MOV 867 348 0
  ***  SCHEDIT  ***
2110 MOV 804 655 0
60   MOV 314 649 0
525  :text howdy planet
120  MOV 75 154 0
40   MOV 100 200 0
40   LEFT 100 200 0
  ***  DV  ***
1562 LEFT 200 200 1
# Calling shelltool with standard sunview
# parameters.
sys /usr/bin/shelltool -Wp 17 376 -Ws 719 516 -WP 1008 108
365  KB q
  ***  EPIC  ***
40   MOV 353 200 0
20   LEFT 818 476 0
  ***  SCHEDIT  ***
1289 :get 244
223  MOV 676 45 0
40   LEFT 676 45 0
333  CENTER 676 45 0
83   :get 2114
223  MOV 676 105 0
40   LEFT 676 105 0
327  CENTER 676 105 0
66   :write 2kram
45   KB r
78   :qu
```

The Test Engineer's Virtual User

A key to the success of AUTOmate in a test environment is its ability to be "program independent" and to become a "virtual user." AUTOmate is "program independent" because it receives low level input data (events) from the applications. These input events are basically raw SunView[3] events which can be used by any SunView application. The sequence of events recorded, however, is only useful for the application from which they were recorded. AUTOmate becomes a *virtual user* by placing these events to drive the application(s) in the same input queue where real mouse and keyboard events are sent (refer to Figure 1 and accompanying discussion on the unified input solution). Therefore, as far as applications are concerned, an AUTOmate session is no different than a user sitting at the display and providing the input directly. This capability allows the test engineer to retain full control of the applications at all times. AUTOmate provides for user interaction at any point during a playback session by pausing AUTOmate, which allows for easy enhancements and maintenance of existing test file since the test engineer can stop the playback, record the new functionality test, and then continue the playback of the original test file.

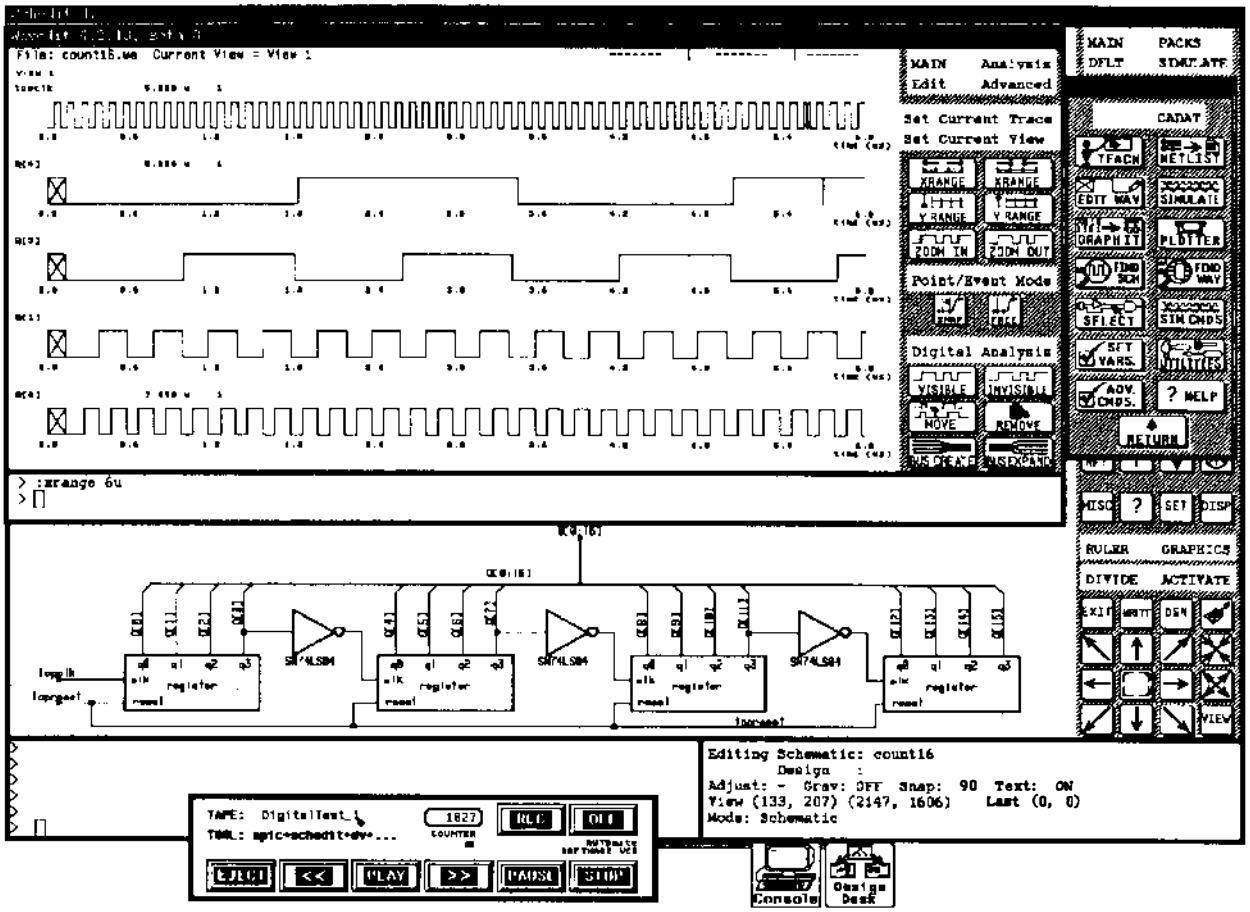


Figure 3. AUTOmate driving ECAE tools

3. SunView is a trademark of Sun Microsystems Inc.

Output Verification

AUTOmate can be used to help monitor the integrity of the software as it progresses through the test suites, or just to check the final results produced by the software at the end of a test run. As the test engineer designs a test suite, (s)he can create check point files by making the software write its internal database to the disk and then add control flow structures, system calls, or AUTOmate commands to the log file so that automatic notification of problems would be reported.

Since AUTOmate allows for system calls and control flow structures in the log file, the tester can use this facility to integrate a bit-mapped verification system to its test suites. Bearing in mind the size of bit mapped files, compaction programs also can be used to help relieve the demand that these files place on disk usage.

The following excerpt of a simple AUTOmate log file illustrates how a test engineer can create a test file to determine the state of a program at a certain point, pause AUTOmate if an error condition is discovered, and display the error message(s). In essence, the test engineer can now add whatever functionality is required to achieve the desired diagnostic effects in the log file.

```
.
.
.
# Save contents of the internal schematic data base into a file
1090 :write check1
# call the diff program with a check point and compare the current
# results with a previous "known good" results.

syst diff check1.sch ../verify/check1.sch > check1.diff

# If the return status from diff is not 0 (potential errors occurred)
# pause the current session of AUTOmate, and popup a window with
# a message and another window with the differences between the
# two files.

if $status != 0
        pauseAUTOmate
        popup 'Problems with SCHEDIT in file >> check1.sch <<'
        showfile check1.diff
else
#       No errors were found; remove temporary files
        syst rm check1.sch check1.diff
endif
.
.
.
```

Reusability

Besides expanding our capabilities in the testing arena, AUTOmate allows for the reusability of our previous test suites using command scripts. This is accomplished because AUTOmate is basically a virtual user and therefore no modifications to our command scripts are necessary to incorporate them to our new testing environment.

Bug Tracking

A future use of AUTOmate will provide bug tracking and reporting. Using this tool will enable us to find out exactly the steps necessary to duplicate and/or understand a problem reported by our customers. Besides problems discussed below, an issue arises when local data, used by our customers, is not sent along with a log file.

Future plans suggest a compact version of AUTOmate to be shipped with a product license. AUTOmate could then be enabled with a special command and the customer, field engineer, or marketing technical support specialist could recreate the customer problem. If the customer recreates the problem, then the resulting log file could be sent to Customer Service Support for analysis. Here is where a problem can arise. In our EDA environment, it is quite possible that local versions of symbol libraries, menus, color maps, and other utilities may be employed at the customer site. If those items are not somehow included with the log file, analysis of the problem will be difficult if not impossible because duplication of the problem (by playing the log file with our tools at the Customer Service Center) would not occur. A solution to this will require further thought.

General Limitations

Ironically, perhaps the greatest weakness of AUTOmate comes from the very feature that makes it so powerful: the capturing of low level input data. In order to playback a recorded session, all the EDA tools used during the recording session must be set up in the same exact manner as when the log file was first recorded. Although AUTOmate has built in features which help the test engineer restore an environment in which the test file was created, even this may not always be sufficient since dependencies of search path setting and other variables could cause the EDA tools to not function as required. What we really need is a method to take a snapshot of the entire operating environment to recreate when replaying a tape file. This suggests the capture of large amounts of data and can be problematic in implementation. Future development, however, must address this facet in order to function as a true bug tracking and customer support device.

A minor problem arises when transferring a test file between hardware platforms running at different speeds, such as in the case of transfer from a 10mip workstation to a 3mip workstation. Whenever a file is recorded on a faster workstation than the one used for playback, the test engineer must use AUTOmate's slowdown features or the applications will not keep pace with the rate of commands sent by AUTOmate from a tape file.

AUTOmate is sensitive to the location of items on a menu. Changes in their location can cause a test not to work properly, but with minor work using AUTOmate's record and edit capabilities, the test file can be repaired and run to completion.

Finally, although AUTOmate filters out most unwanted events, the recorded session can still result in a huge file due to the mouse movements. This data, however, can be edited out (by hand or by use of a compact program) so that unnecessary mouse movements are deleted, while their timing dependencies are added to the remaining events.

In Summary

The AUTOmate project maintains the goal of improving, refining, and expanding the scope and quality of software product quality assurance. Through applications in the area of bug tracking, customer support, and regular software shakedown, AUTOmate allows the test engineer and support personnel to incorporate a variety of diagnostic tools to poke and prod at software while it is actually running in its native domain. By creating a new program we were able to unload from the ECAE tools the need to encode and decode data from the log files, gained the flexibility to add control structures, system calls, and AUTOmate commands in the log file, and the ability to test an integrated environment.

References & Notes

[1] In chapter 8 of the *SunView System Programmer's Guide*, Release 4.0, Revision A of May 9th, 1988, Sun Microsystems acknowledges the possibility of extending their notifier to provide record and replay mechanisms, but asserts that such functionality is not supported by their current interface. *SunView System Programmer's Guide 4.0, Rev. A*, Chapter 8 – *Advanced Notifier Usage*, p.91. Sun Microsystems, Inc., Mountain View, California. 1988.

BGG: A Testing Coverage Tool[1]

Mladen A. Vouk and Robert. E. Coyle[2]

North Carolina State University
Department of Computer Science, Box 8206
Raleigh, N.C. 27695-8206

Abstract

BGG, **B**asic **G**raph **G**eneration and Analysis tool, was developed to help studies of static and dynamic software complexity, and testing coverage metrics. It is composed of several stand-alone modules, it runs in UNIX environment, and currently handles static and dynamic analysis of control and data flow graphs (global, intra-, and inter-procedural data flow) for programs written in full Pascal. Extension to C is planned. We describe the structure of BGG, give details concerning the implementation of different metrics, and discuss the options it provides for treatment of global and inter-procedural data flow. Its capabilities are illustrated through examples.

Biographical Sketches

Mladen A. Vouk received B.Sc. and Ph.D. degrees from University of London (U.K.) in 1972 and 1976 respectively. From 1980 to 1984 he was Chief of the Programming Languages Department at the University Computing Centre, University of Zagreb, Yugoslavia. He is currently Assistant Professor of Computer Science at North Carolina State University. His areas of interest include software reliability and fault-tolerance, software testing, numerical software, and mixed language programming.

Dr. Vouk is the current secretary of Working Group 2.5 (Numerical Software) of the Technical Committee 2 of the International Federation for Information Processing. He is a member of IEEE, ACM, and Sigma Xi.

Robert E. Coyle received the B.S. degree in mechanical engineering from Fairleigh Dickinson University, Teaneck, N.J. in 1983 and is currently completing the M.S. in computer science at North Carolina State University. He is Software Group Leader at the Teletec Corporation working on the design and development of real-time, embedded telecommunications systems. In the past worked in telecommunications systems development with the Singer Company and IBM. His research interests are in software engineering, particularly metrics, testing technology, and software reliability.

Mr. Coyle is a member of the IEEE Computer Society, and the Association for Computing Machinery.

---

[1]Research supported in part by NASA Grant No. NAG-1-983
[2]Teletec Corporation, Raleigh, N.C.

BGG: A Testing Coverage Tool

Mladen A. Vouk and Robert. E. Coyle[1]

North Carolina State University
Department of Computer Science, Box 8206
Raleigh, N.C. 27695-8206

I. Introduction

Software testing strategies and metrics, and their effectiveness, have been the subject of numerous research efforts (e.g. comparative studies by Nta88, Cla85, Wei85, and references therein). Practical testing of software usually involves a combination of several testing strategies in hope that they will supplement each other. The question of which metrics should be used in practice in order to guide the testing and make it more efficient remains largely unanswered, although several basic coverage measures seem to be generally considered as the minimum that needs to be satisfied during testing.

Structural, or "white-box", approaches use program control and data structures as the basis for generation of test cases. Examples include branch testing, path testing [Hen84, Woo80] and various data flow approaches [Hec77, Las83, Rap83, Fra88]. Functional, or "black-box", strategies rely on program specifications to guide test data selection [e.g. How80,87, Dur84]. Some of the proposed strategies combine features of both functional and structural testing as well as of some other methods such as error driven testing [Nta84].

Statement and branch coverage are regarded by many as one of the minimal testing requirements; A program should be tested until every statement and branch has been executed at least once, or has been identified as unexecutable. If the test data do not provide full statement and branch coverage the effectiveness of the employed testing strategy should be questioned. Of course, there are a number of other metrics which can provide a measure of testing completeness. Many of these are more sophisticated and more sensitive to the program control and data flow structure than statement or branch coverage. They include path coverage, domain testing, required elements testing, TER_n ($n \geq 3$) coverage, etc. [How80, Hen84, Whi80, Nta84 and reference therein].

---

[1]Teletec Corporation, Raleigh, N.C.

The simplest data-flow measure is the count of definition-use pairs or tuples [Her76]. There are several variants of this measure. More sophisticated measures are p-uses, all-uses, and du-paths [Fra88, Nta88], ordered data contexts [Las83], required pairs [Nta84,88], and similar. The data-flow based metrics have been under scrutiny for some time now as potentially better measures of the testing quality than control-flow based metrics [e.g. Las83, Rap83, Fra88, Wey88]. However, one recent study [Zei88] indicates that most of the data-flow metrics may not be sufficiently complete for isolated use, and that in practice they should be combined with control-flow based measures.

Over the years a number of software tools for measuring various control and data flow properties and coverage of software code have been reported [e.g. Ost76 (DAVE), Fra86 (ASSET), Kor88]. Unfortunately, in practice these tools are either difficult to obtain, or difficult to adapt to specific languages and research needs, or both. To circumvent that, and also gain better insight into the problematics of building testing coverage tools, we have developed a system for static and dynamic analysis of control and data flow in software.

The system, BGG (Basic Graph Generation and Analysis system), was built as a research tool to help understand, study, and evaluate the many software complexity and testing metrics that have been proposed as aids in producing better quality software in an economical way. BGG allows comparison of coverage metrics and evaluation of complexity metrics. It can also serve as a support tool for planning of testing strategies (e.g. stopping criteria), as well as for active monitoring of the testing process and its quality in terms of the coverage provided by the test cases used. Section II of the paper provides an overview of the BGG system structure and functions. Section III gives details concerning the implementation of various metrics and of handling local, global and inter-procedural data flow. Section IV illustrates the tool capabilities through examples.

II. Structure and Functions

A simplified top level diagram of BGG is shown in Figure 1. BGG is composed of several modules which can be used as an integrated system, or individually given appropriate inputs, to perform static and dynamic analyses of control and data flow in programs written in Pascal. The tool currently handles full Berkeley Pascal[1] with one minor exception. The depth of the "with" statement nesting is limited to one. The extension to greater depth is simple and will be implemented in the next version of the system. BGG runs in UNIX environment. Its

[1]Standard UNIX compiler, pc.

implementation under VM/CMS is planned together with its extension to analysis of programs written in the C language. BGG itself is written in Pascal, C and UNIX C-shell script.

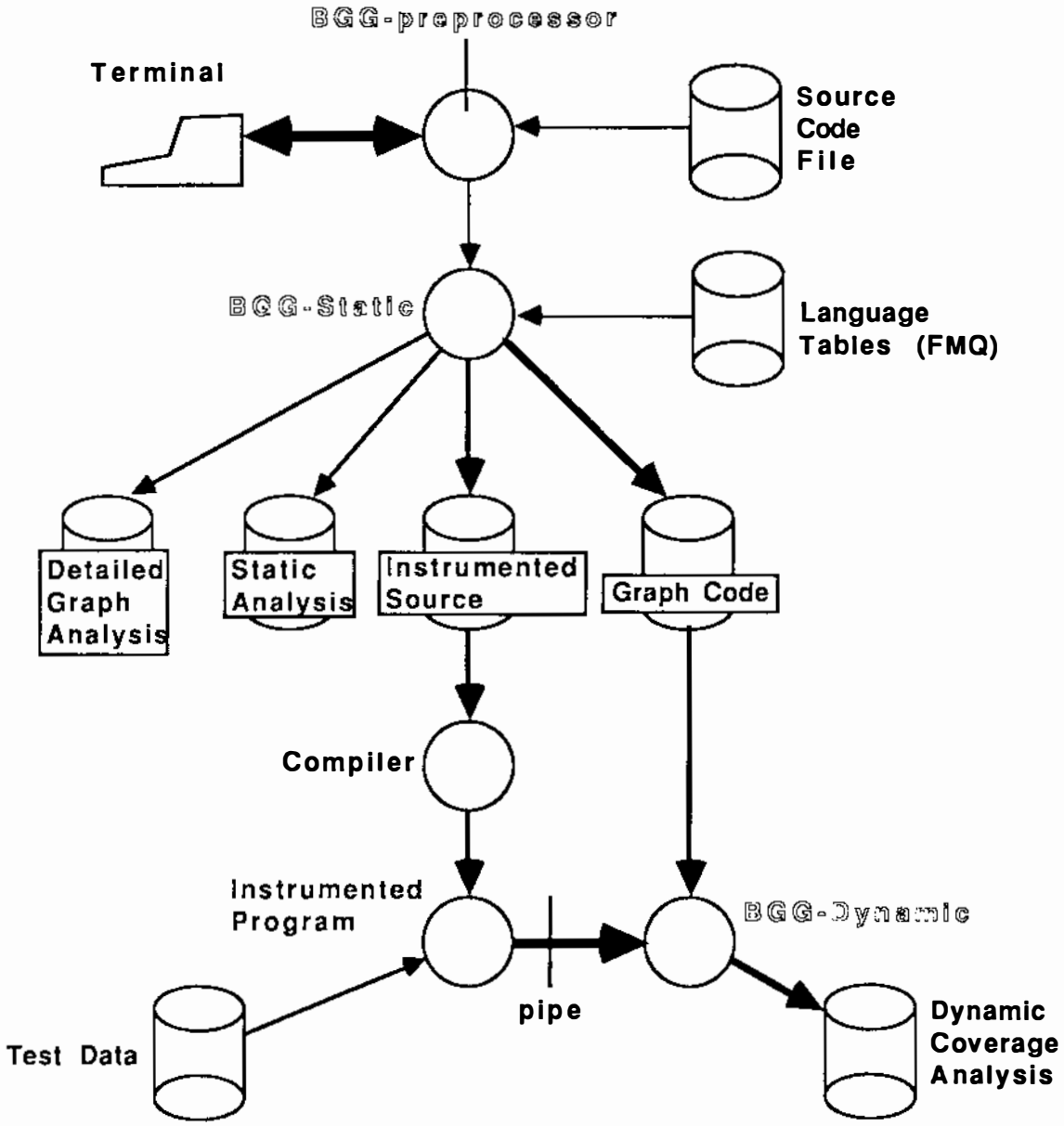


Figure 1. Schematic diagram of the information flow in the BGG system of tools.

BGG pre-processor provides the user interface when the tool is used as an integrated system. It also performs some housekeeping chores (checks for file existence, initializes appropriate language tables and files, etc.), and prepares the code for processing by formatting it and stripping it of comments. The language tables are generated for the system once, during the system installation, and then stored. The front-end parsing is handled through the FMQ generator [Mau81, Fis88].

This facility also allows for relatively simple customization of the system regarding different programming languages and language features. Also, each of the BGG modules has a set of parameters which can be adjusted to allow analyses of problems which may exceed the default values for the number of nodes, identifier lengths, nesting depth, table sizes, etc.

Pre-processed code, various control information and language tables are used as input to the BGG-Static processor. This processor constructs control and data-flow graphs, and performs static analysis of the code. These graphs are the basis for all further analyses. Statistics on various metrics and control-flow and data-flow anomalies, such as variables that are used but never defined etc, are reported. BGG-Static also instruments the code for dynamic execution tracing.

When requested, BGG executes the instrumented code with provided test cases and analyzes its dynamic execution trace through BGG-Dynamic. The dynamic analysis output contains information (by procedures and variables) about the coverage that the test cases provide under different metrics.

When instrumenting code BGG inserts a call to a special BGG procedure at the beginning of each linear code block. It also adds empty blocks to act as collection points for branches. The instrumentation overhead in executable statements is roughly proportional to the number of linear blocks present in the code. In our experience this can add between 50% and 80% to the number of executable lines of code. The run-time tracing overhead for the instrumented programs is proportional to the number of linear blocks of code times the cost of the call to the BGG tracing procedure. The latter simply outputs information about the block and the procedure being executed.

The raw run-time tracing information may be stored in temporary files, and processed by BGG-Dynamic later. However, often the amount of raw tracing information is so large that that it becomes impractical to store it. BGG-Dynamic can then accept input via a pipe and process it on-the-fly. Because BGG-Dynamic analyses may be very memory and CPU intensive, particularly in the case of data-flow metrics, interactive testing may be a slow process. Part of the problem lies in the fact that BGG is still a research tool and was not optimized. We expect that the next version of BGG will be much faster and more conservative in its use of memory. It will permit splicing of information from several short independent runs, so that progressive coverage can be computed without regression runs on already executed data.

Currently BGG computes the following static measures: counts of local and global symbols, lines of code (with and without comments), total lines in executable control graph nodes, linear blocks of code, control graph edges and graph nodes, branches, decision points, paths (the maximum number of iterations through loops can be set by the user), cyclomatic number, definition and use counts for each variable, definition-use (du) pair counts, definition-use-redefinition (dud) chain counts, count of definition-use paths, average definition-use path lengths, p-uses, c-uses, and all-uses. Dynamic coverage is computed for definition-use pairs, definition-use-redefinition chains, p-uses, c-uses and all-uses. Definition-use path coverage and path coverage for paths that iterate loops k times (where k can be set by user) will be implemented. There are several system switches which allow selective display and printing of the results of the analyses.

III. Graphs and Metrics

Control and data flow graphs

Each linear block of Pascal code is a node in a graph. A linear code sequence is a set of simple Pascal statements (assignments, I/O, and procedure/function calls), or it is a decision statement of an iterative or conditional branching construct. When a linear block is entered during execution all of its statements are guaranteed to be executed. Decision statements are always separated out into single "linear blocks". Procedure/function calls are treated as simple statements which use or define identifiers and/or argument variables. A linear block node has associated with it a set describing variables defined in it, and a set describing variables used in it. Also attached to each node is the node execution information.

In each Pascal statement all identifiers for simple local and global variables, named constants defined using CONST, and all built-in Pascal functions are considered. Built in functions are treated as global identifiers. For the purpose of the definition-use analyses, explicit references to elements of an array are treated as references to the array identifier only. Similarly, references to variables pointed to by pointers are currently reduced to references to the first pointer in the pointer chain. An extension that will differentiate between a stand-alone use of a pointer (e.g. its definition or use in a pointer expression), and use of a pointer, or a pointer chain, for de-referencing of another variable, will be implemented in the next version of the tool. Input/output statement identifiers (function names) are considered used, while their argument variables are used (e.g. write, writeln) or defined (e.g. read, readln). The file identifier is treated as a simple variable (defined for input, used for output).

Calls to functions or procedures are treated as local statements which use the procedure/function[1] identifier. In the case of function calls this use is preceded by one or more definitions of the function identifier in the called function itself. This definition is propagated to the point of call, where a single definition of the function identifier is then followed by its local use. From the point of view of the calling procedure, the actual argument variables are either used once, or defined once, or both used and defined once (in that order), depending on whether the corresponding parameter is used (any number of times), defined (any number of times), or used and defined (in any order) in the procedure that is called. Definitions are returned only if the corresponding parameter is a **var** parameter.

The point of call ordering: used-defined, for **var** parameters used and defined in any order, was chosen as a warning mechanism for programmers that have access to analyses of their own code but may not have access to the analyses, or the actual code, of the procedures they call. The idea is to impress on the programmers that the variable may be used in the invoked unit, and therefore they should be careful about what they send to it because the definition may not mask an undefined argument variable, an illegal value, etc. The way we handle procedure arguments permits a limited form of inter-procedural data flow analysis, and offers a more realistic view of the actual flow of information through the code. It also means that the code for the called procedures must be available for BGG to analyze. An alternative is not to use this option, but use the defensive approach of assuming that every argument variable of a **var** parameter is always used and then defined.

A global variable that has been only used in a called procedure, or used in the procedures called within the called procedure, is reported as used at the point of call. A global variable that has been only defined in the called procedure, or deeper, is reported defined at the point of call. However, a global variable that has been used and defined (in any order) in the called procedure, or in any procedure called within the called procedure to any depth, is reported as defined and then used at the point of call. The reason global variables are treated differently from procedure arguments is to highlight global variable definition in the called procedure(s) by making it visible as a definition-use pair at the point of call. Again, it is a form of warning to the programmers that the underlying procedures have changed a global variable value, may have re-used this value, and in turn may have (if the definition was erroneous) affected values of some, or all, the parameter values passed back to the point of call.

[1]From here on, we use term "procedure" to mean procedure or function, unless a distinction has to be made between the two.

All procedure parameters are assumed to be defined (pseudo-defined) on entry. Global variables used in a procedure are also pseudo-defined on entry. Parameters and global variables set in a procedure or function are assumed used (pseudo-used) on exit. The actual use and definition of completely global variables, and locally global variables, is fully accounted for in each procedure in which they occur as far as their uses and re-definitions are concerned. On return to the calling procedure, any global variables that have been used or defined in the called procedure are reported as single uses and/or definitions of that global variable at the point of call, however, pseudo-uses enabled within a procedure are not reported back to the point of call. The tool has options that allow different treatment of global variables (e.g. pseudo definitions and uses can be switched off), and selective display of the analyses of only some functions and procedures.

Iteration constructs are treated as linear blocks containing the decision statement followed (while, for), or preceded (repeat), by the subgraphs describing the iteration body. Conditional branching constructs (if, case of) consist of decision nodes followed by two or more branch subgraphs. All decision points are considered to have p-uses (edge associated uses) as defined in [Fra88].

Metrics

Some of the static metrics that BGG currently computes are less common or are new and require further explanations.

By default, path counts are computed so that each loop is traversed once. However, definition-use-redefinition chain counts (see below) force on some loops one more iteration in addition to the first traversal. User may change the default number of iterations through a loop through a switch (one value for all loops). Cyclomatic number is computed in the usual way [McC76]. Implemented data flow visibility of all language constructs and variables is such that full definition-use coverage implies full coverage of executable nodes (and in turn full statement coverage) [e.g. Nta88]. BGG computes c-uses, p-uses, and all-uses according to definitions given in [Fra88].

Definition-use-(re)definition, d(u)d, chains are data-flow constructs defined in [Vou84]. It is one of the metrics we are currently evaluating for sensitivity and effectiveness in software fault detection. A d(u)d is a linear chain composed of a definition followed by a number of sequential uses, and finally by a re-definition of a variable. The basic assumptions behind this metric are a) the longer a d(u)d chain is the more complex is the use of this variable, and b) the more one re-defines a variable the more complex its data-flow properties are. The first property is measured

through d(u)d length (see below), the second property is measured by counting d(u)d's. An additional characteristic of d(u)d chains is that they are cycle sensitive and, for those variables where they are present, they force at least two traversals of loops within which a variable is defined. However, full d(u)d coverage does not imply full du-pair coverage. The d(u)d metric is intended as a supplementary measure to other definition-flow metrics.

Definition of a du-path can be found, for example, in [Fra88, Nta88]. A single du-pair may have associated with it one or more du-paths from the definition to that use. We augment the count of du-paths and du-pair counts with measures of du-path lengths. The assumption is that, from the standpoint of complexity (and hence affinity to errors), it is not only the count of du-paths that is important, but also the length of each path. A definition which is used several times, perhaps in physically widely separated places in the program, requires more thought and may be more complex to handle than one that is defined and the used only once, or for the first time. For each du-path we compute length by counting the number of edges covered in reaching the paired use. For every variable we also compute an average length over all du-pairs and du-paths associated with it. In a similar manner we define d(u)d-length as the number of use-nodes between the definition and redefinition points of the chain. Average d(u)d-length is the d(u)d-length accumulated over all d(u)d pairs divided by the number of d(u)d's. We use d(u)d-lengths to augment d(u)d-counts.

We also distinguish between linear (or acyclic) d(u)d's and loop generated, or cyclic, d(u)d's. Cyclic d(u)d's are those where the variable re-defines itself or is re-defined in a cyclic chain. All cyclic constructs are potentially more complex than the linear ones. Comparison is difficult unless the loop count is limited, or looping is avoided, in which case cyclic structures lend themselves to comparison with acyclic ones through unfolding. If iterative constructs are regarded only through du-pairs, many cycles may not be detected since all du-pairs might be generated by going around a loop only once. On the other hand, for a cyclic d(u)d to be generated, a second pass through a loop is always required. However, if there are no definitions of a variable within a loop then the loop would not be registered by d(u)d constructs belonging to that variable. When a variable is only used (or not used at all) within a loop, its value is loop invariant and loop does not add any information that the variable can (legally) transmit to other parts of the graph.

BGG also has facility for computing concentration (or density) of du-paths and d(u)d-paths through graph nodes. We believe that graph (code) sections that show high du-chain and d(u)d-chain node densities may have a higher probability of being associated with software faults than low density regions.

IV. Examples

The examples given in this section derive from an ongoing project where BGG is being used to investigate static and dynamic complexity properties of multi-version software, multi-version software fault profiles, and effectiveness and efficiency of different testing strategies. We are using two sets of functionally equivalent numerical programs for these studies. One set consists of 6 Pascal programs (average size about 500 lines of code) described in [Vou86], the other set consists of 20 Pascal programs (average size about 2,500 lines of code) described in [Kel88].

```
158   function fptrunc(x: real): real;
159
160
161   const
162         largevalue = 1.0e18;
163   var
164         remainder: real;
165         power: real;
166         bigpart: real;
167         term: real;
168   begin
169         remainder := abs(x);
170         if remainder > largevalue then
171               fptrunc := x
172         else begin
173               power := 1.0;
174               while power < remainder do
175                     power := power * 10.0;
176               bigpart := 0.0;
177
178               while remainder > maxint do begin
179
180                     while power > remainder do
181                           power := power / 10.0;
182               term := power * trunc(remainder / power);
183               remainder := remainder - term;
184               bigpart := bigpart + term;
185               end;
186
187               remainder := trunc(remainder) + bigpart;
188
189               if x < 0.0 then
190                     fptrunc := -remainder
191               else
192                     fptrunc := remainder;
193         end;
194   end;
```

Figure 2. Code section for which analysis is shown in Figures 3 and 4

Figure 2 shows a section of the code from program L17.3 of the 6-program suite. Figure 3 illustrates the output that static analysis processor "BGG-Static" offers in the file "Static Analysis" for the same procedure.

Outputs like the one shown in Figure 3 provide summary profile of each local and global symbol found in the code. How many times it was defined (or pseudo-defined), used (or pseudo-used), how many du-pairs it forms, how many d(u)d chains, etc. This static information can be used to judge the complexity of procedures, or the complexity of the use of individual variables. In turn, this information may help in deciding which of the variables and procedures need additional attention on the part of the programmers and testers.

Figure 4 illustrates the detailed node, parameter, and global variable information available in the file labelled "Detailed Graph Analysis" in Figure 1. Figure 4 is annotated (bold text) to facilitate understanding. We see that all parameters (e.g. X), global variables (e.g. TRUNC), and built-in functions (e.g. ABS) are pseudo-defined on entry. The parenthesized number following a capitalized identifier is its number in the symbol table. Note that there are empty nodes, inserted by BGG, which act as collection points for branches (e.g. Block #17). Because FPTRUNC was defined in several places in the code, it is pseudo-used on exit from the function (in Block #18). Note also that built-in function ABS is treated as a global variable, and its parameters are used only (because BGG does not have insight into its code), but the situation is different in the case of locally defined procedures.

For example, Figure 5 shows another section of the code in which procedure ADJUST calls a local function FPMOD (line 285) which, in turn (not shown), calls function FPFLOOR, which then calls function FPTRUNC. The details of the static analysis of the first ADJUST node where the call chain begins are shown in Figure 6. Output lines relevant to the discussion are in bold. Note that FPTRUNC is global from the point of view of ADJUST and is therefore pseudo-defined on entry. The same is true for FPMOD and FPFLOOR. All three are reported as defined and then used in line 285. For two of them the use actually occurs at a deeper level, in function FPMOD for FPFLOOR, and in function FPFLOOR for FPTRUNC. The definitions occur in functions themselves, e.g. for FPTRUNC it occurs in FPTRUNC itself. All these underlying definitions and uses are propagated back to ADJUST.

Procedure: FPTRUNC

Local Symbol Name	Def Ct	Use Ct	DU Ct	DUD Ct	Cyclic DUD Ct	Acyclic DUD Ct	DU Len	DUD Len	Cyc Len	Acyc Len	Subpath Ct	Puse Ct	Cuse Ct
FPTRUNC	3	1	3	0	0	0	1.00	0.00	0.00	0.00	303		
X	1	3	3	0	0	0	1.88	0.00	0.00	0.00	13	2	2
LARGEVALUE	1	1	1	0	0	0	1.00	0.00	0.00	0.00	13	2	0
REMAINDER	3	9	14	9	2	7	4.14	5.33	4.50	5.57	12	12	8
POWER	3	6	16	6	3	3	2.61	2.83	3.00	2.67	16	10	11
BIGPART	2	2	4	4	2	2	1.00	1.00	1.00	1.00	8	0	4
TERM	1	2	2	2	2	0	1.50	2.00	2.00	0.00	4	0	2
Global Symbol Name													
ABS	1	1	1	0	0	0	1.00	0.00	0.00	0.00	13	0	1
MAXINT	1	1	1	0	0	0	1.67	0.00	0.00	0.00	13	2	0
TRUNC	1	2	2	0	0	0	1.40	0.00	0.00	0.00	13	0	2
Procedure Summary:	17	28	47	21	9	12	2.62	3.48	2.67	4.08	108	28	33

Total local symbols:	7
Total global symbols:	3
Total lines (in executable nodes):	27
Total lines (in non-empty xqt nodes):	25
Total blocks:	18
Total non-empty blocks:	16
Total edges:	22
Total branches:	10
Total decisions:	5
Total paths:	13
Cyclomatic number:	6

Figure 3. Static analysis of the procedure shown in Figure 2.

```
Procedure FPTRUNC (193)
  Parameter X (194)

  Global TRUNC (28) used                          global variable
  Global MAXINT (55) used                         global variable
  Global ABS (7) used                             built in function
  Global FPTRUNC (193) defined
---------------------------------------- ••• ----------------------------------------
Procedure # 8 : FPTRUNC

Block # 1
Nodetype: NOT FOR                                 not a for-loop
Predecessor list: none
Successor list: 2
Block start at line: 168
Block ends at line: 169
Definition-use list for this block:
  ABS (7) is defined in line # 168                pseudo-def
  MAXINT (55) is defined in line # 168            pseudo-def
  TRUNC (28) is defined in line # 168             pseudo-def
  LARGEVALUE (195) is defined in line # 168       constant
  X (194) is defined in line # 168                pseudo-def
  X (194) is used in line # 169
  ABS (7) is used in line # 169                   (1)
  REMAINDER (196) is defined in line # 169

Block # 2
Nodetype: NOT FOR
Predecessor list: 1
Successor list: 4 3
Block start at line: 170
Block ends at line: 170
Definition-use list for this block:
  LARGEVALUE (195) is used in line # 170
  REMAINDER (196) is used in line # 170

Block # 3
Nodetype: NOT FOR
Predecessor list: 2
Successor list: 18
Block start at line: 170
Block ends at line: 171
Definition-use list for this block:
  X (194) is used in line # 171
  FPTRUNC (193) is defined in line # 171
---------------------------------------- ••• ----------------------------------------
Block # 17                                        empty collector node
Nodetype: NOT FOR
Predecessor list: 15 16
Successor list: 18
Block start at line: 193
Block ends at line: 193
Definition-use list for this block: empty

Block # 18                                        end node
Nodetype: NOT FOR
Predecessor list: 3 17
Successor list: none
Block start at line: 194
Block ends at line: 194
Definition-use list for this block:
  FPTRUNC (193) is used in line # 194             pseudo-use
```

(1) ABS is a built in function and is treated as a global identifier only

Figure 4. Elements of the detailed node analysis

```
274   procedure adjust(var p: point);
275
276
277   var
278         twopi, piover2: real;
279   begin
280         twopi := pi * 2;
281         piover2 := pi / 2;
282
283         begin
284
285               p.long := fpmod(p.long, twopi);
286
287               p.lat := fpmod(p.lat, twopi);
288               if p.lat > pi then
289                     p.lat := p.lat - twopi;
290
291               if p.lat > piover2 then
292                     p.lat := pi - p.lat
293               else if p.lat < -piover2 then
294                     p.lat := -pi - p.lat;
295         end;
296   end;
```

Figure 5. Code section for which static analysis is shown in Figure 6.

Of course, variables strictly local to FTPRUNC, such as "**remainder**" (see also Figures 2, 3 and 4), do not show at the point of call to FPMOD in ADJUST. It is obvious that global data flow can add considerably to the mass of definitions, uses, and other constructs a programmer has to worry about. Nevertheless, we believe that it is a good practice to make this information available so that the full implication of a call to a procedure can be appreciated.

BGG provides coverage information on the program level, and on the procedure level. Figure 7 illustrates output from the dynamic coverage processor "BGG-Dynamic", delivered in the "Dynamic Coverage Analysis" output file, for function FPTRUNC and a set of 103 test cases. In the example some of the output information normally delivered by BGG has been turned off, e.g. all-uses.

For each procedure BGG-Dynamic first outputs a summary of branch coverage information: the block number, statement numbers encompassed by the block, the number of times the block was executed, and the execution paths taken from the block (node). For example, node 5 in Figure 7 was executed 724 times, 6 times to node 3, and 721 times to node 7. Branches which have not been executed show up as having zero executions.

```
Procedure ADJUST (214)
  Parameter P.LONG (216) used
  Parameter P.LONG (216) defined
  Parameter P.LAT (217) used
  Parameter P.LAT (217) defined

  Global WRITELN (35) used
  Global FPFLOOR (200) used
  Global ABS (7) used
  Global MAXINT (55) used
  Global TRUNC (28) used
  Global FPTRUNC (193) used
  Global FPMOD (203) used
  Global PI (107) used
  Global FPMOD (203) defined
  Global FPFLOOR (200) defined
    Global FPTRUNC (193) defined                    procedure in scope, global variable
------------------------------ ••• ------------------------------
Procedure # 13 : ADJUST

Block # 1
Nodetype: NOT FOR
Predecessor list:
Successor list: 2
Block start at line: 279
Block ends at line: 287
Definition-use list for this block:
  PI (107) is defined in line # 279
  FPMOD (203) is defined in line # 279
    FPTRUNC (193) is defined in line # 279          pseudo-definition
  TRUNC (28) is defined in line # 279
  MAXINT (55) is defined in line # 279
  ABS (7) is defined in line # 279
    FPFLOOR (200) is defined in line # 279          pseudo-definition
  WRITELN (35) is defined in line # 279
  P.LAT (217) is defined in line # 279
  P.LONG (216) is defined in line # 279
  P (215) is defined in line # 279
  PI (107) is used in line # 280
  TWOPI (218) is defined in line # 280
  PI (107) is used in line # 281
  PIOVER2 (219) is defined in line # 281
  TWOPI (218) is used in line # 285          <---------- Beginning of the list of
  P.LONG (216) is used in line # 285                     visible definitions and uses
  FPMOD (203) is defined in line # 285                   for line 285 from Figure 5.
                                                 calls FPFLOOR
    FPFLOOR (200) is defined in line # 285       calls FPTRUNC
  FPTRUNC (193) is defined in line # 285
  WRITELN (35) is used in line # 285
  FPFLOOR (200) is used in line # 285            actually used in FPMOD
  ABS (7) is used in line # 285
  MAXINT (55) is used in line # 285
  TRUNC (28) is used in line # 285
    FPTRUNC (193) is used in line # 285          actually used in FPFLOOR
    FPMOD (203) is used in line # 285            actually used in ADJUST
  P.LONG (216) is defined in line # 285      <---------- List for line 285 ends
  TWOPI (218) is used in line # 287
  P.LAT (217) is used in line # 287
  FPMOD (203) is defined in line # 287
  FPFLOOR (200) is defined in line # 287
  FPTRUNC (193) is defined in line # 287
  WRITELN (35) is used in line # 287
  FPFLOOR (200) is used in line # 287
  ABS (7) is used in line # 287
  MAXINT (55) is used in line # 287
  TRUNC (28) is used in line # 287
  FPTRUNC (193) is used in line # 287
  FPMOD (203) is used in line # 287
  P.LAT (217) is defined in line # 287       <---------- Block #1 ends
```

Figure 6. Elements of the detailed static analysis of the procedure shown in Figure 5.

Proc. #: 8 Proc/Func Name: FPTRUNC

Block	Statements	Executions	Branches(Executions)
1	168 - 169	721	2(721)
2	170 - 170	721	3(0) 4(721)
3	170 - 171	0	18(0)
4	172 - 173	721	5(721)
5	174 - 174	724	6(3) 7(721)
6	174 - 175	3	5(3)
7	176 - 177	721	8(721)
8	178 - 178	721	9(0) 13(721)
9	178 - 179	0	10(0)
10	180 - 180	0	11(0) 12(0)
11	180 - 181	0	10(0)
12	182 - 185	0	8(0)
13	186 - 188	721	14(721)
14	189 - 189	721	15(66) 16(655)
15	189 - 190	66	17(66)
16	191 - 192	655	17(655)
17	193 - 193	721	18(721)
18	194 - 194	721	

Non-empty blks exec/total non-empty blks = 12 / 16 = 75.00%
Lines in executed nodes/total lines in executable nodes = 18 / 25 = 72.00%
Branches exec/total branches = 6 / 10 = 60.00%
Legend: Ct - Total static count, Def - Definitions, Use - Uses, DU - Definition-Use Pairs, DUD - Definition-Use-(Re)Definition Chains
PecentX - percentage of the static count executed with the current set of test cases, Puse - Predicate-Uses, Cuse - Computational Uses

Local Symbol Name	Def Ct	Use Ct	DU Ct	PercentX	DUD Ct	PercentX	Puse Ct	PercentX	Cuse Ct	PercentX
FPTRUNC	3	1	3	66.67	0	0.00	0	0.00	3	66.67
DU pairs not executed:										
	Defined: 171		Used: 194							
X	1	3	3	66.67	0	0.00	2	100.00	2	50.00
DU pairs not executed:										
	Defined: 168	Used: 171								
LARGEVALUE	1	1	1	100.00	0	0.00	2	50.00	0	0.00
P-uses not executed:										
	Defined: 168		Used: 170		Successor: 3					
REMAINDER	3	9	14	42.86	9	22.22	12	33.33	8	37.50
DU pairs not executed:										
	Defined: 183	Used: 183								
	Defined: 183	Used: 182								

Figure 7. Dynamic analysis of the procedure in Figure 2 based on 103 test cases.

Defined: 183 Used: 180
Defined: 183 Used: 187
Defined: 183 Used: 178
Defined: 169 Used: 183
Defined: 169 Used: 182
Defined: 169 Used: 180

DUD chains not executed:

Defined: 183 Used: 178 Used: 180 Used: 180 Used: 182 Used: 183 Defined: 183
Defined: 183 Used: 178 Used: 180 Used: 182 Used: 183 Defined: 183
Defined: 183 Used: 178 Used: 187 Defined: 187
Defined: 169 Used: 170 Used: 174 Used: 174 Used: 178 Used: 180 Used: 180 Used: 182 Used: 183 Defined: 183
Defined: 169 Used: 170 Used: 174 Used: 174 Used: 178 Used: 180 Used: 182 Used: 183 Defined: 183
Defined: 169 Used: 170 Used: 174 Used: 178 Used: 180 Used: 180 Used: 182 Used: 183 Defined: 183
Defined: 169 Used: 170 Used: 174 Used: 178 Used: 180 Used: 182 Used: 183 Defined: 183

P-uses not executed:

Defined: 183 Used: 180 Successor: 12
Defined: 183 Used: 180 Successor: 11
Defined: 183 Used: 178 Successor: 13
Defined: 183 Used: 178 Successor: 9
Defined: 169 Used: 180 Successor: 12
Defined: 169 Used: 180 Successor: 11
Defined: 169 Used: 178 Successor: 9
Defined: 169 Used: 170 Successor: 3

POWER 3 6 16 18.75 6 16.67 10 30.00 11 9.09

DU pairs not executed:

Defined: 181 Used: 181
Defined: 181 Used: 182
Defined: 181 Used: 182
Defined: 181 Used: 180
Defined: 175 Used: 175
Defined: 175 Used: 181
Defined: 175 Used: 182
Defined: 175 Used: 182
Defined: 175 Used: 180
Defined: 173 Used: 181
Defined: 173 Used: 182
Defined: 173 Used: 182
Defined: 173 Used: 180

DUD chains not executed:

Defined: 181 Used: 180 Used: 181 Defined: 181
Defined: 181 Used: 180 Used: 182 Used: 182 Used: 180 Used: 181 Defined: 181
Defined: 175 Used: 174 Used: 175 Defined: 175

Figure 7 (continued 1). Dynamic analysis of the procedure in Figure 2 based on 103 test cases.

Defined: 175 Used: 174 Used: 180 Used: 181 Defined: 181
Defined: 173 Used: 174 Used: 180 Used: 181 Defined: 181
P-uses not executed:
Defined: 181 Used: 180 Successor: 12
Defined: 181 Used: 180 Successor: 11
Defined: 175 Used: 180 Successor: 12
Defined: 175 Used: 180 Successor: 11
Defined: 175 Used: 174 Successor: 6
Defined: 173 Used: 180 Successor: 12
Defined: 173 Used: 180 Successor: 11

BIGPART	2	2	4	25.00	2	0.00	0	0.00	4	25.00

DU pairs not executed:
Defined: 184 Used: 184
Defined: 184 Used: 187
Defined: 176 Used: 184
DUD chains not executed:
Defined: 184 Used: 184 Defined: 184
Defined: 176 Used: 184 Defined: 184

TERM	1	2	2	0.00	1	0.00	0	0.00	2	0.00

DU pairs not executed:
Defined: 182 Used: 184
Defined: 182 Used: 183
DUD chains not executed:
Defined: 182 Used: 183 Used: 184 Defined: 182

Global Symbol Name	Def Ct	Use Ct	DU Ct	PercentX	DUD Ct	PercentX	Puse Ct	PercentX	Cuse Ct	PercentX
ABS	1	1	1	100.00	0	0.00	0	0.00	1	100.00
TRUNC	1	2	2	50.00	0	0.00	0	0.00	2	50.00

DU pairs not executed:
Defined: 168 Used: 182

MAXINT	1	1	1	100.00	0	0.00	2	50.00	0	0.00

P-uses not executed:
Defined: 168 Used: 178 Successor: 9

Procedure Summary:	17	28	47	38.30	18	16.67	28	39.29	33	30.30

Figure 7 (continued 2). Dynamic analysis of the procedure in Figure 2 based on 103 test cases.

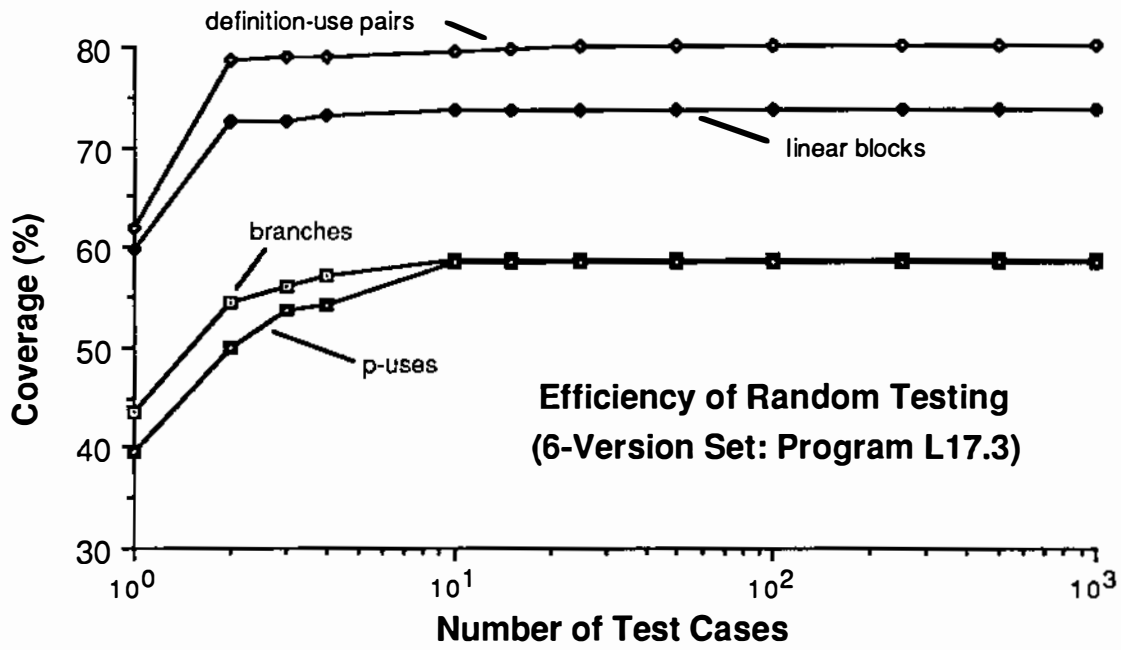


Figure 8. Coverage observed during random testing of a program from the 6-version set.

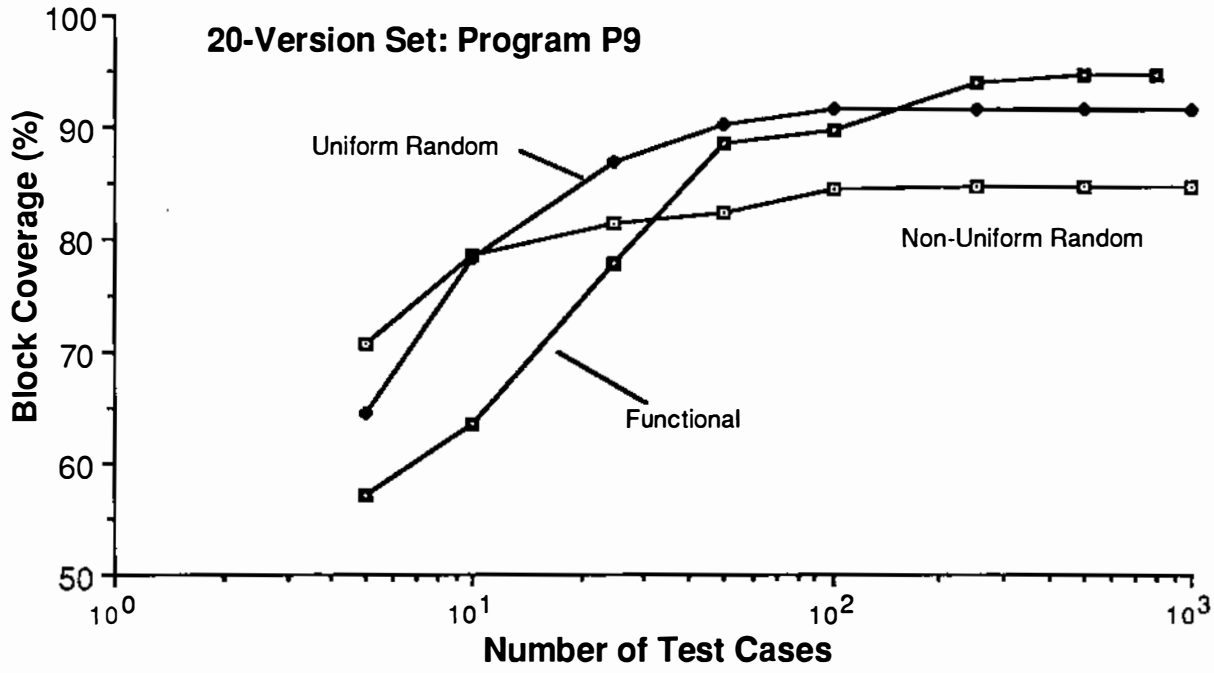


Figure 9. Comparison of linear block coverage observed for two random testing profiles and a functional data for a program out of the 20-version set.

The branch table is followed by a summary of coverage by metrics: coverage for non-empty blocks (blocks that have not been inserted by BGG), lines of code within executable nodes, and branch coverage. This is followed by coverage for data flow metrics by symbol name. The static definition, use, du-pair, d(u)d, p-use, etc. counts for a variable are printed along with the information on its the dynamic coverage expressed as percentage of the executed static constructs. For each identifier, this is followed by a detailed list and description of constructs that have not been executed (e.g. du-pairs or p-uses). Execution coverage output tables can be printed in different formats (e.g. counts of executed constructs, rather than percentages), and with different content (e.g. all-uses).

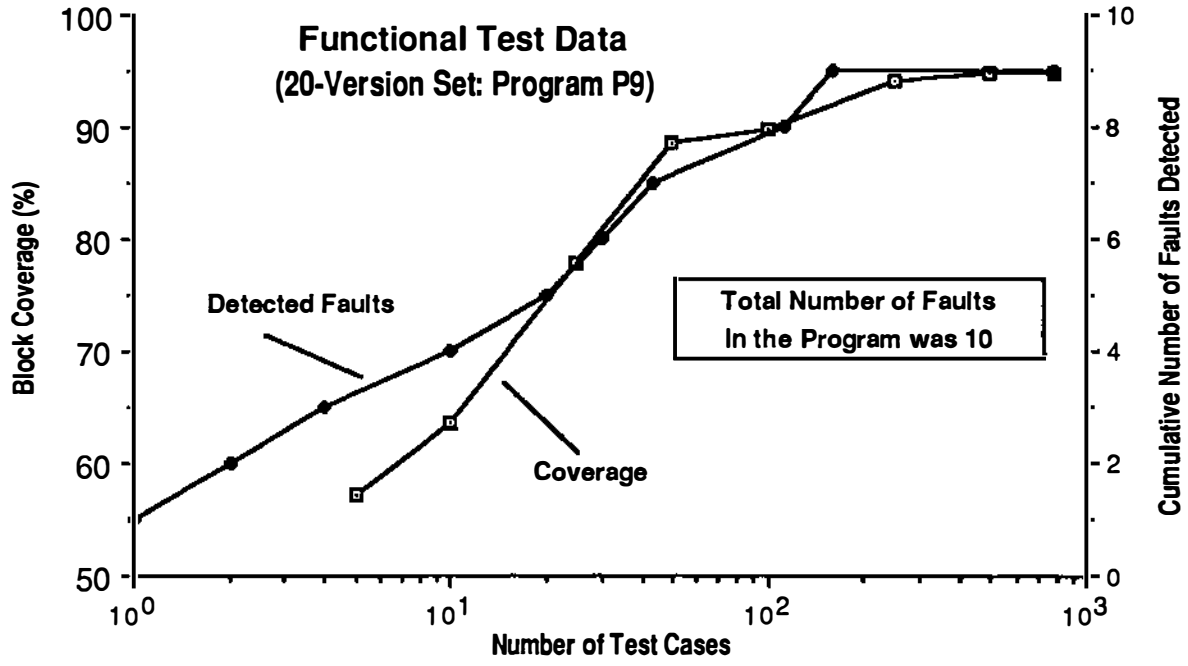


Figure 10. Linear block coverage and fault detection efficiency observed for program P9 of the 20-version set with functional test cases.

BGG can also be used to obtain coverage growth curves for a particular test data set. Figures 8 and 9 illustrate this. They show some of the coverage growth curves we have observed with random and functional (designed) test cases for the program L17.3 of the 6-version set using BGG described here, and for a program P9 from the 20 version set using an early version of the system.

It is interesting to note that both figures show coverage that follows an exponential growth curve and reaches a plateau extremely quickly. For the smaller program (Figure 8, about 600 lines of

code) metrics reach saturation already after about 10 cases, while for the larger program (20-version set, about 2,500 lines of code) this happens after about 100 cases. There is also a marked difference in the initial slope and the plateau level obtained with different testing profiles.

Once the coverage is close to saturation for a particular testing profile, its fault detection efficiency drops sharply. This is illustrated in Figure 10 where we plot the coverage provided by the functional testing profile shown in Figure 9, and the cumulative number of different faults detected using these test cases. Out of the 10 faults that the code contained, 9 were detected with the functional data set used within the first 160 cases.

It is clear that apart from providing static information on the code complexity, and dynamic information on the quality of test data in terms of a particular metric, BGG can also be used to determine the point of diminishing returns for a given data set, and help in making the decisions on when to switch to another testing profile or strategy.

V. Summary

We have described a research tool that computes and analyses control and data flow in Pascal programs. We plan to extend the tool into C language. We have found BGG to be very useful in providing information for code complexity studies, in directing execution testing by measuring coverage, and as a general unit testing tool that provides programmers with information and insight that is not available through standard UNIX tools such as the pc compiler, or the pxp processor. We are currently using BGG in an attempt to formulate coverage based software reliability models by relating code complexity, testing quality (expressed through coverage), and the number of faults that have been discovered in the code.

References

[Cla85] L.A. Clarke, A. Podgurski, D. Richardson, and S. Zeil, "A comparison of data flow path selection criteria," in Proc. 8th ICSE, pp 244-251, 1985.

[Dur84] J.W. Duran and S.C. Ntafos, "An Evaluation of Random Testing," IEEE Trans. Software Eng., vol. SE-10, pp. 438-444, 1984.

[Fis88] C.N. Fisher and R.J. LeBlanc, *Crafting a compiler*, The Benjamin/Cummings Co., 1988.

[Fra86] P.G. Frankl and E.J. Weyuker, "A data flow testing tool," in Proc. SoftFair II, San Francisco, CA, pp 46-53, 1985.

[Fra88] P.G. Frankl and E.J. Weyuker, "An applicable family of data flow testing criteria," IEEE Trans. Soft. Eng., Vol. 14 (10), pp 1483-1498, 1988.

[Hen84] M.A. Hennell, D. Hedley and I.J. Riddell, "Assessing a Class of Software Tools", Proc. 7th Int. Conf. Soft. Eng., Orlando, Fl, USA,pp 266-277, 1984.

[Hec77] M.S. Hecht, *Flow Analysis of Computer Programs*, Amsterdam, The Netherlands: North-Holland, 1977.

[Her76] P.M. Herman, "A data flow analysis approach to program testing," Australian Comput. J., Vol 8(3), pp 92-96, 1976.

[How80] W. E. Howden, "Functional Program Testing," IEEE Trans. Software Eng., Vol. SE-6, pp.162-169,1980.

[How87] W.E. Howden, "Functional Program Testing and Analysis", McGraw-Hill Book Co., 1987.

[Kel88] J. Kelly, D. Eckhardt, A. Caglayan, J. Knight, D. McAllister, M. Vouk, "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results", Proc. FTCS 18, pp 9-14, June 1988.

[Kor88] B. Koren and J. Laski, "STAD - A system for testing and debugging: user perspective," Proc. Second Workshop on Software Testing, Verification, and Analysis, Banff, Canada, Computer Society Press, pp 13 - 20, 1988.

[Las83] J.W. Laski and B. Korel, "A Data-Flow Oriented Program Testing Strategy", IEEE Trans. Soft. Eng., Vol. SE-9, pp 347-354, 1983.

[Mau81] J. Mauney and C.N. Fischer, "FMQ -- An LL(1) Error-Correcting-Parser Generator", User Guide, University of Wisconsin-Madison, Computer Sciences Technical Report #449, Nov. 1981.

[McC76] T. McCabe, "A Complexity Measure," IEEE Trans. Soft. Eng., Vol. SE-2, 308-320, 1976.

[Nta84] S.C. Ntafos, "On Required Element Testing", IEEE Trans. Soft. Eng., Vol. SE-10, pp 793-803, 1984.

[Nta88] S.C. Ntafos, "A Comparison of Some Structural Testing Strategies", IEEE Trans. Soft. Eng., Vol. SE-14 (6), pp 868-874, 1988.

[Ost87] L.J. Osterweil and L.D. Fosdick, "DAVE - a validation, error detection and documentation system for FORTRAN programs," Software - Practice and Experience, Vol. 6, 473-486, 1976.

[Rap85] S. Rapps and E.J. Weyuker, "Selecting software test data using data flow information," IEEE Trans. Soft. Eng., Vol. SE-11(4), pp 367-375, 1985.

[Vou84] M.A. Vouk and K.C. Tai, "Sensitivity of definition-use data-flow metrics to control structures", North Carolina State University, Department of Computer Science, Technical Report: TR-85-01, 1985.

[Vou86] M.A. Vouk, D.F. McAllister, and K.C. Tai, "An Experimental Evaluation of the Effectiveness of Random Testing of Fault-tolerant Software", Proc. Workshop on Software Testing, Banff, Canada, IEEE CS Press, 74-81, July 1986.

[Wei85] M.D. Weiser, J.D. Gannon, and P.R. McMullin, "Comparison of structured test coverage metrics," IEEE Software, Vol 2(2), pp 80-85, 1985.

[Wey88] E.J. Weyuker, "An empirical study of the complexity of data flow testing", Proc. Second Workshop on Software Testing, Verification, and Analysis, Banff, Canada, Computer Society Press, pp 188-195, 1988.

[Whi80] L.J. White and E.J. Cohen, "A Domain Strategy for Computer Program Testing", IEEE Trans. Soft. Eng., Vol. SE-6, pp 247-257, 1980.

[Woo80] M. R. Woodward, D. Hedley, and M. A. Hennell, "Experience With Path Analysis and Testing of Programs," IEEE Trans. Software Eng., vol.SE-6, pp.278-286, 1980.

[Zei88] S.J. Zeil, "Selectivity of data flow and control flow path criteria", Proc. Second Workshop on Software Testing, Verification, and Analysis, Banff, Canada, Computer Society Press, pp 216-222, 1988.

HARDWARE TESTING AND SOFTWARE ICS

Daniel Hoffman
University of Victoria
Department of Computer Science
P.O. Box 1700
Victoria, B.C., Canada V8W 2Y2

ABSTRACT

Despite substantial research on testing methods and support tools, little help is available for the tester in the field. Proposals for test case selection are typically too costly to apply in practice. Test tools are complex and limited in availability. There is particularly little support for the testing of individual modules, as opposed to complete software systems.

In contrast, hardware testing is a mature field where sophisticated testing techniques and tools are in widespread use. This paper presents a practical approach to testing based on adapting chip testing concepts to software module regression testing. Test cases are defined formally using a language based on module traces, and a software tool is used to automatically generate test programs that apply the cases. The testing approach, language and program generator are described in detail.

BIOGRAPHICAL SKETCH

Dr. Daniel Hoffman received the B.A. degree in mathematics from the State University of New York, Binghamton, in 1974, and the M.S. and Ph.D. degrees in computer science in 1981 and 1984, from the University of North Carolina, Chapel Hill. From 1974 to 1979 he worked as a commercial programmer/analyst. He is currently an Assistant Professor of Computer Science at the University of Victoria. His research area is software engineering, emphasizing software specification and testing.

HARDWARE TESTING AND SOFTWARE ICS

Daniel Hoffman*
University of Victoria
Department of Computer Science
P.O. Box 1700
Victoria, B.C., Canada V8W 2Y2

INTRODUCTION

The fundamental goal of our research is to improve system quality and reduce maintenance costs through systematic module regression testing. While considerable attention is given to testing during software development, this is not the only time testing is required. As Brooks points out:

> As a consequence of the introduction of new bugs, program maintenance requires far more system testing per statement written than any other programming. Theoretically, after each fix one must run the entire test bank of test cases previously run against the system, to ensure that it has not been damaged in an obscure way. In practice such *regression testing* must indeed approximate this theoretical ideal, and it is very costly [1, pg. 122].

Successful regression testing depends on the ability to economically maintain and execute a large set of test cases along with the production code. While system testing is usually emphasized, module testing is also important. It is difficult to thoroughly test a module in its production environment, just as it is difficult to effectively test a chip on its production board. IEEE testing standards [2] emphasize the benefits of testing software components, not just complete systems.

In the sections to follow we discuss related work, provide the essential terminology, compare current approaches to hardware and software testing, and describe the testing approach we have developed by viewing modules as "software ICs".

RELATED WORK

The focus of our work is test case execution — relatively little has been published in this area. The DAISTS system [3] focuses on module testing and describes test cases using sequences of calls. Given a formal algebraic specification of the module under test, DAISTS automatically determines the correct behavior for a given test. Panzl [4] reports on regression testing of Fortran subroutines. He presents a test case description language and a program to automatically execute the cases, monitoring actual versus expected behavior. Stucki [5] describes techniques and tools for inserting assertions in the code under test, each expressing an invariant relationship between variable values. A test input that falsifies one of the assertions indicates a bug in the code. Our work is most similar to the DAISTS work, which goes further than ours by providing a test oracle, but offers little for the common situation when an algebraic specification is unavailable.

* Research supported by the Natural Sciences and Engineering Research Council of Canada under grant A8067.

Considerable work has been reported in the area of test case selection. In *black-box* testing [6, 7], the tests are based on the requirements of the program. *White-box* testing [8, 9] uses the internal structure of the program to select appropriate test cases. Our approach supports the use of both black-box and white-box testing. Because we emphasize the module interface and follow Howden's proposals for *functional* testing [6], our test case selection strategy might be considered black-box. However, it is white-box to the extent that we also base test cases on the module implementation.

TERMINOLOGY

Modules and Interfaces

Following Parnas [10], we define a *module* as a programming work assignment, and a *module interface* (hereafter just *interface*) as the set of assumptions that programmers using the module are permitted to make about its behavior. An *interface specification* is a statement, in some form, of these assumptions. We view a module as a black box, accessible only through a fixed set of *access programs*. The *syntax* of the specification states the names of the access programs, their parameter and return value types, and the names of the *exceptions* that each access program may generate. Any constants or types provided by the module are also described. The *semantics* of the specification state, for each access program call, the situations in which the call is legal, and the effect that invoking the call has on the legality and return values of other calls. By convention, in access program names we use the prefix `s_` (set) to indicate calls which set internal module values and `g_` (get) to indicate calls which retrieve those values. These ideas are illustrated on a simple table module which is used as an example throughout this paper.

Program name	Inputs	Outputs	Exceptions
s_init			
g_space		integer	
s_addsym	string		maxlen legsym tblfull
s_delsym	integer		notlegid
g_legsym	string	boolean	
g_legid	integer	boolean	
g_sym	integer	string	notlegid
g_id	string	integer	notlegsym

Figure 1. Symbol Table Interface Syntax

The Symbol Table (*symtbl*) module maintains a set of symbol/identifier pairs. The syntax is shown in Figure 1; the semantics are described informally below and formally elsewhere [11]. Up to S unique symbols may be stored, each up to N characters in length.

Each symbol has a unique integer identifier, assigned by *symtbl* from the range $[0, S-1]$. The access programs are divided into three groups.

1. `s_init` initializes the module and must be called before any other call. `g_space` returns the amount of available space in the table.
2. `s_addsym(`s`)` adds the symbol s to the table, signaling `maxlen` if s is longer than N, `legsym` if s is already in *symtbl*, and `tblfull` if `g_space` $= 0$. `s_delsym(`i`)` deletes the symbol with identifier i and signals `notlegid` if no symbol in the table has that identifier.
3. `g_legsym` and `g_legid` are the characteristic predicates of the set of legal symbols and the set of legal identifiers, respectively. `g_sym(`i`)` returns the symbol with identifier i, signaling `notlegid` if `g_legid(`i`)` is false; `g_id(`s`)` returns the identifier for symbol s, signaling `notlegsym` if `g_legsym(`s`)` is false.

Testing

A *test case* is an input/output pair, where an "input" is a stimulus to the module and an "output" is some observable behavior. *Scaffolding* [1, pg. 148] has long been used in modular software development. Consider a module M. A *driver* is a program written to call access programs provided by M. A *stub* is a program that serves as a substitute for an access program called by M. Drivers and stubs support top-down and bottom-up testing, respectively. For software testing, we adapt two important concepts well-known in hardware testing [12]. *Controllability* refers to the ease with which an arbitrary stimulus may be applied to a module or a submodule. *Observability* refers to the ease with which the required behavior of a module or submodule may be observed.

HARDWARE TESTING VERSUS SOFTWARE MODULE TESTING

In order to adapt hardware testing techniques to software, it is essential to have a clear understanding of both the similarities and the differences between the two testing tasks. For simplicity, we will use *module* to refer to software and *chip* to refer to hardware.

Similarities

1. Testing is expensive and plays a critical role in quality assurance.
2. Test costs can be significantly reduced by *design for testability*, where controllability and observability are influential factors. Although this influence is often ignored by software developers, hardware designers are keenly aware of it [12].
3. The decomposition of systems into modules has a significant effect on testability. In hardware, the high cost of each additional pin places significant limitations on the communication paths between chips. In software, there is no such physical limitation. There is, for example, no time or space cost in communication through global variables. Indeed, the introduction of global variables often provides performance improvements. Yet, experience has shown that uncontrolled module interactions significantly reduce testability, as well as comprehensibility and maintainability. In other words, software has a limitation, due to intellectual manageability, akin to the pin limitations in chip design.

Differences

We see five differences between hardware and software, relevant to the testing task.

1. As is typical in manufacturing, the hardware fabrication process is expensive and faulty, and the purpose of testing is to determine whether each fabricated chip is an accurate copy. The correctness of the designed circuit is established during an earlier *verification* step and is taken as given during testing. In software, the situation is reversed. The copying process has negligible cost and is nearly flawless, and testing is used to determine the correctness of the original. This difference between hardware and software is profound and must not be ignored.
2. In hardware development, exhaustive testing is a well-known technique, effective because the input domain is often relatively small ($\leq 2^{20}$) and tests can be run at hardware speeds. In software, exhaustive testing is extremely rare, because the input domains are astronomical in size.
3. Although fabrication techniques and circuit designs may be extremely complex, the functional specification of a chip is typically relatively simple, and is often formally defined independently of the mask. For software modules, the service offered is typically complex and is rarely described precisely and completely, except by the source code itself.
4. On chips, it is difficult to access circuits except through the pin interface — insertion of probes is expensive and may alter the chip's behavior. In contrast, it is straightforward to insert testing code in modules without altering module behavior.
5. Since large numbers of exact copies of a given chip are often manufactured, substantial investment in a test harness is justifiable. A module test harness will be used far less often, and even then on variants of the module, not exact copies. So, the module test harness must be inexpensive to build and to change.

MODULE TESTING STRATEGY

We have based our approach to module testing on the similarities and differences described above. In general, we have found that the basic principles of hardware testing apply well to software while the specific techniques do not.

Basic Principles

Our approach to testing is based on the following principles.

- Systematic module regression testing

  Often tests are developed during or after implementation and are discarded shortly after acceptance. It is important to plan testing early in development and to deliver and maintain tests along with the production code. A system should not be considered complete until both production code and test code are complete. Without testing in place, the system may appear finished, but it is a maintenance problem in disguise.

  In successful testing, the savings from detected errors must exceed the costs of generating, maintaining, executing and evaluating the tests. While this statement appears obvious, it is often ignored. The focus in the literature has been test input generation. The cost of test maintenance is rarely mentioned. *Test oracles* are normally assumed

as given, and schemes are proposed that generate large quantities of test inputs. In practice, generation of expected outputs would be done manually, at significant cost.

- Design for testability

  For testing to be cost effective, testability must be an influential design criteria. Critical to testability is the ability to isolate the module under test. It is typically difficult to thoroughly test a module M while it is installed in a production system. M's access programs are often not directly accessible. If M is a general-purpose module, some of its access programs may not be called at all in a particular production system. Errors in other modules may appear to be errors in M. Conversely, errors in M may be masked by errors in other modules. In short, when M is running in its production environment, controllability and observability are significantly reduced. Using test scaffolding, M may be tested in complete isolation from the production environment. In practice, modules are best tested with a mixture of scaffolding and production code; the critical tradeoff is between the benefits realized through isolation, and the cost of developing and maintaining the scaffolding.

- Apply automation cost-effectively

  Rather than automating tasks where manual approaches are cost-effective or those where it is unclear how automation can be applied, we focus on clerical tasks performed repeatedly. These tasks are the least costly to automate and savings on tasks that are repeated often are, obviously, realized a large number of times. Since regression tests are developed once and run many times, cost-saving efforts are directed towards test case execution and evaluation. Our tests are developed manually, with automated support, and then run fully automatically. We rely on the test programmer's ability to effectively select test cases.

  The automation of software testing has been severely hampered by lack of standardization. Simple standards for module interfaces and test case description can help significantly. These standards need not be industry-wide; the overhead of tool building may be justified even if the standard's use is limited to a single system.

Viewing Modules as Software ICs

When viewing modules as software ICs, we use the following correspondence. As access programs are the means by which module users access the module's services, each access program corresponds to a pin, or closely related set of pins. Each exception corresponds to a an incorrect usage of the chip. To improve controllability and observability, we may add test access programs to the module interface and embed assertions in the module implementation. These correspond to test pins and on-chip test circuitry, respectively. The use of such test code is especially attractive for modules because the code can easily be removed from the production code using standard conditional compilation techniques.

TEST CASE DESCRIPTION

Our test scripts are written in terms of module *traces*: sequences of access program calls on the module. Elsewhere traces have been used in connection with the formal specification method of the same name [13, 14] in which logic assertions are used to characterize module

behavior on *all* possible traces. Although the trace specification method is powerful, considerable skill is required to devise these assertions. It is straightforward, however, to describe module behavior for any *particular* trace. For example, consider the following traces on the *symtbl* module. (When writing traces, we separate adjacent calls with a period.)

```
s_init().s_addsym("cat").g_legsym("cat")
s_init().s_addsym("cat").g_id("dog")
```

In the first trace are calls to initialize *symtbl*, add the symbol `"cat"` and check to see if `"cat"` is a legal symbol. In the second trace, the `g_id` call should generate the exception `notlegsym` because the symbol `"dog"` is not in *symtbl*.

We describe test cases by providing a trace and associating it with some aspect of the required behavior of the module following that trace. We represent a test case as a five-tuple

$$< trace, expexc, actval, expval, type >$$

with the following meanings:

trace
: a trace used to exercise the module.

expexc
: the name of the exception that *trace* is expected to generate (or `noexc` if no exception is expected).

actval
: an expression (typically a get call) to be evaluated after *trace*, whose value is taken to be the "actual value" of the trace.

expval
: the value that *actval* is expected to have.

type
: the data type of *actval* and *expval*.

Below are two test cases, based on the traces described above. In test cases developed solely to do exception checking, the *actval*, *expval*, and *type* fields contain `empty`.

```
<s_init().s_addsym("cat"), noexc, g_legsym("cat"), 1, boolean>
<s_init().s_addsym("cat").g_id("dog"), notlegsym, empty, empty, empty>
```

A test script consists of a list of access program and exception declarations, a list of test cases, and optional global C code. C code, delimited by the symbols "{%" and "%}", may also be embedded in test cases and provides the test programmer with expressions and control structures not supported by PGMGEN. A test script may be viewed as a partial specification for a module, expressing its required behavior under specific circumstances. The purpose of PGMGEN is to generate a driver which will determine whether an implementation satisfies this partial specification.

TEST PROGRAM GENERATION

Although implementing test drivers manually is straightforward, it is also tedious and error-prone, and produces code that is costly to maintain. As a result, test driver generation is a

good candidate for automated support. In this section we briefly describe how PGMGEN accomplishes this task; a more detailed description is available [15]. Initially, code is generated to record exception occurrences. Then, for each test case of the form:

$$< c_1.c_2.....c_N, expexc, actval, expval, type >$$

code is generated to:

```
invoke c1, c2, ..., cN
compare the actual occurrences of exceptions against expexc
if there are any differences
        print a message
else
        if actval ≠ expval
                print a message
        if any exceptions have occurred since cN was invoked
                print a message
update summary statistics
```

Following the last case, code is generated to print summary statistics.

In order to automate driver generation, we have standardized access program invocation and exception signaling as follows. Each access program is implemented as a C function. For each exception, there is a C function of that name, serving as exception handler. When an exception occurs, the module implementation must call the appropriate function. The module user is expected to implement the exception handlers to take whatever action he deems suitable. In a *symtbl* implementation, each set and get call is implemented as a C function and the functions `legsym`, `maxlen`, `notlegid`, `notlegsym`, and `tblfull` are invoked when the corresponding exception is detected. However, the *symtbl* user implements these exception functions. PGMGEN, for example, implements the exception handlers to set flags for monitoring exception occurrences.

EXPERIENCE

Batch and Interactive Drivers

We have used the approach described above for the testing of roughly 50 modules, including most of the modules in the PGMGEN implementation itself [15]. Many of these modules were tested with drivers generated from PGMGEN scripts. Scripts typically contain 25–100 test cases and take 2-4 hours to develop. We have found the scripts significantly easier to develop and maintain than the manually generated drivers used previously. In particular, the scripts are roughly an order of magnitude shorter than the generated drivers. We have also made use of drivers that are interactive, as opposed to the batch drivers produced by PGMGEN. Our interactive drivers prompt the user for an access program name and parameter values, invoke the access program and, for a get call, display the return value. Because these drivers are simple and standardized, we have been able to generate them automatically. The availability of both batch and interactive drivers for most modules has shown an important difference between the two. Batch drivers are effective for regression testing, where large numbers of

test cases must be run repeatedly, but are awkward for debugging, where the behavior of one test case determines what other test cases are interesting. Conversely, interactive testers are good for debugging, but poor for regression testing.

Design for Testability

Our experience has shown that testability varies widely across modules. In terms of controllability and observability, we have found two major obstacles to testability.

- Interaction with the environment

  Modules that interact with their environment through means *other* than access program calls and return values often have controllability or observability problems. Modules supporting keyboard input present controllability problems, unless a convenient means of storing and "replaying" keystroke sequences is available. For modules doing cursor positioning or line editing, observability is typically a problem as well. Modules generating screen output also cause observability problems. Even when software access to the screen contents is provided, the access method is often complex, and may have side-effects on the screen contents or cursor position. Modules generating file output present some observability problems. Although files are accessible to software, file comparison utilities must be used, usually at some additional cost.
- Inaccessible code or data

  Some modules have internal code or data structures that should be tested but is not readily accessible through the module interface, i.e., that is not readily controlled or observed. For example, in *symtbl* the symbols and identifiers are accessible but not conveniently. It may be necessary to call `g_legsym` once for every identifier between 0 and N-1 to determine the complete table contents — even if the table is known to contain a single symbol.

We have used four techniques to overcome these obstacles to testability.

- Alternate decomposition

  Often a change in module decomposition can significantly improve testability. For example, many keyboard input modules do parsing as well as raw input processing. As a result, the parsing code suffers unnecessarily from the controllability problems inherent in keyboard handling. Implementing separate modules for parsing and keyboard input improves the testability of the parsing code. To reduce maintenance costs, we have based our module decompositions on the information hiding principle [16]. Happily, we have found that decomposition according to information hiding usually reduces test costs as well.
- Scaffolding

  Our primary use of scaffolding has been the drivers described above. We have also made frequent use of stubs to monitor the calls made by the module under test on other modules, and have programs available to automatically generate stub skeletons. We have also written keystroke capture and screen dump programs that reduce the cost of testing keyboard and screen modules. It has become clear that controllability and observability depend not only on the module under test but also on the scaffolding available.

- Test access programs

  In cases where a module is difficult to test through the access programs on its interface, we have added access programs for test purposes. In these cases, the module has two interfaces: the "public" interface and the "testing" interface. We have frequently implemented the test access program **g_dump**, which displays the contents of the module's internal data structures. With **g_dump**, testing is done in two stages: (1) invoke the set access programs and use **g_dump** to see if the data structures have been altered appropriately and (2) invoke the get access programs to see if they correctly return the values shown by **g_dump**. With simple modules, this separation of concerns is useful — with complex ones, it is invaluable.
- Embedded assertions

  Embedding executable boolean tests in the module implementation provides a second way to test code not readily accessible through the module interface. We have made heavy use of code to check for access program misuse, as defined by the exceptions specified in the module interface. Integration testing proceeds more smoothly with modules that protect their internal data against corruption and signal attempts at misuse. We have explored the use of assertions that express internal data structure invariants. We believe that much more use could be made of such assertions.

CONCLUSIONS AND FUTURE WORK

We have argued that software testing can benefit by incorporating concepts from hardware testing. Most critical is the commitment to developing software systems from relatively simple, systematically tested modules. Specifically, we have proposed systematic regression testing, design for testability and automated testing, and have presented practical techniques for achieving these goals.

We have found little use for the specific techniques of hardware testing. This is not surprising because hardware testing focuses on manufacturing (copying) flaws,while software testing focuses on human flaws in the original program. The hardware /software comparison has led us to seriously question the suitability of structural software testing. Put simply, structural testing seems better suited to detecting manufacturing errors than human errors. While this is an intuitive argument against structural testing, arguments in favor of structural testing are intuitive as well. It is well-known that trivial bugs may remain undetected by even the most rigorous structural testing.

We see three areas for future work.

1. To make software testing economical, a much better understanding of its costs is needed — especially the cost of module regression testing and of integration testing of systematically tested modules.
2. Because thorough testing often requires large numbers of similar test cases, test scripts quickly become long and repetitive. We have begun work on test case generation [17] — much more remains to be done.
3. The use of assertions in testing has significant potential. We would like to be able to write assertions such as $x >= y + 100$ and be notified if the assertion is ever false during a test run. Instrumenting a program to perform this notification is conceptually simple,

but too tedious to do manually. Stucki [5] proposed a tool for this task. While it is unclear whether this feature was ever implemented it certainly is not widely available now.

In conclusion, if the object oriented paradigm is to become an industry standard, systematic module regression testing will be a critical quality assurance technique.

REFERENCES

[1] F.P. Brooks. *The Mythical Man-Month.* Addison-Wesley, 1975.

[2] *IEEE Standard for Software Unit Testing.* Soft. Eng. Tech. Comm. of the IEEE Computer Society, May 1987.

[3] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, July 1981.

[4] D.J. Panzl. A language for specifying software tests. In *Proc. AFIPS Natl. Comp. Conf.*, pages 609–619, AFIPS, 1978.

[5] L.G. Stucki. *New Directions in Automated Tools for Improving Software Quality*, pages 80–111. Volume Vol. II: Program Validation, Prentice-Hall, 1977.

[6] W.E. Howden. Functional program testing. *IEEE Trans. Soft. Eng.*, SE-6(2):162–169, March 1980.

[7] J.W. Duran and S.C. Ntafos. An evaluation of random testing. *IEEE Trans. Soft. Eng.*, SE-10(4):438–444, July 1984.

[8] J.C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, September 1975.

[9] W.E. Howden. Reliability of the path analysis testing strategy. *IEEE Trans. Soft. Eng.*, SE-2(3):208–215, September 1976.

[10] D.L. Parnas and P.C. Clements. A rational design process: how and why to fake it. *IEEE Trans. Soft. Eng.*, SE-12(2):251–257, February 1986.

[11] D.M. Hoffman. Practical interface specification. *Software - Practice and Experience*, 19(2):127–148, February 1989.

[12] E.J. McCluskey. *Logic Design Principles.* Prentice-Hall, 1986.

[13] Wolfram Bartussek and David L. Parnas. Using traces to write abstract specifications for software modules. In *Information Systems Methodology*, pages 211–236, Springer-Verlag, 1978. Proc. 2nd Conf. European Cooperation in Informatics, October 10-12, 1978.

[14] D.M. Hoffman and R. Snodgrass. Trace specifications: methodology and models. *IEEE Trans. Soft. Eng.*, 14(9), 1988.

[15] D.M. Hoffman. A CASE study in module testing. In *Proc. Conf. Software Maintenance (accepted for publication)*, IEEE Computer Society, October 1989.

[16] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[17] D.M. Hoffman and C. Brealey. Module test case generation. In *Proc. 3rd Symp. on Software Testing, Analysis, and Verification (accepted for publication)*, ACM SIGSOFT, 1989.

A Software Process Management Approach to Quality and Productivity

Herb Krasner and Mike Pore
Lockheed Software Technology Center

Abstract. The overt management of certain large-scale software development processes will help control technical performance in terms of a product's quality and development productivity.

Key words/phrases: process management, process models, quality framework, productivity framework.

Herb Krasner is currently a research manager and principal investigator in the Lockheed Software Technology Center, leading a research program in advanced technologies for software process management. He was previously the leader of the Empirical Studies of Software Design Project at the Microelectronics and Computer Technology Corporation (MCC) Software Technology Program. Prior to that he held positions as: a software methodologist at Harris Government Systems, Professor of Computer Science at Clemson University, and independent contract systems analyst. He has 17 years of experience in software engineering research and its management in both industrial and academic settings, method/environment/tool development, practical large system engineering, and university teaching. His current research interests include: (1) the collection and analysis of realistic data about problems facing users/developers of large systems, (2) design team collaboration mediated by computer-based groupware, (3) interdisciplinary, "systems" approaches to software design, (4) decision and information-based models of design process, (5) AI applied to software engineering, (6) software (and system) development models, methods, and technology, and (7) theoretical foundations of design. He is active in professional societies, ACM and IEEE Computer Society, and has served as Chairman of several international conferences and as the Director of the ACM Scholastic Student Programming Contest. He has published and presented his work in many forums. Krasner received an MS in computer science from the University of Missouri.

Mike Pore is a research scientist at the Lockheed Software Technology Center. He has experience in aerospace applications of mathematics, statistics, and systems engineering. His current research interests include process modeling and evaluation and software process metric development and analysis. He is also an Adjunct Associate Professor in the Department of Management Sciences and Information Systems at the University of Texas at Austin. Pore received a PhD in mathematics from Texas Tech University.

Address questions about this article to either author at Lockheed, O/96-10, B/30E, P.O. Box 17100, Austin, TX 78760-7100. E-mail addresses:
krasner@stc.lockheed.com *or* pore@stc.lockheed.com

A Software Process Management Approach to Quality and Productivity

Herb Krasner and Mike Pore
Lockheed Software Technology Center

INTRODUCTION: THE CONCERNS OF SOFTWARE PROCESS MANAGEMENT

Many have speculated as to why software projects seem more difficult to manage then other types (e.g., bridge construction). The main reasons include the conceptual nature of the materials used, the mathematical bases of digital computer programs, the invisibility of the partially completed system, the development "tooling up" requirements, the typical structural disconnect between systems and the world they model, the interdisciplinary knowledge needs of unprecedented systems, the inherent complexity caused by uncertainty and change, and the lack of standard methods and piece parts. Large software projects are currently organized and managed according to guidelines adapted from other fields (cf., Metzger 1972; Burrill and Ellsworth 1980). The goal of these is to control the responsibilities, authority, and communications of a project team and not necessarily to coordinate the flow of technical information needed for effective technical performance.

Project management support systems are traditionally concerned with quantitative support, such as calculating cost, determining schedule, ensuring accurate accounting, and reporting variances. This quantitative support is primarily product-oriented, and focuses on *what* must be produced, *when* it will be produced, and *how much* it will cost, thereby attempting to reduce project management to the simplicity of a job-shop scheduling problem in which the product is produced mechanically by passing from one station to the next. From this point of view, the production process is prescribed and predetermined. Traditional project management support systems are not usually concerned with qualitative support, such as the quality of the product, the productivity of people, or the efficiency of the software development process. However, qualitative support is primarily process-oriented, and focuses on *who* is involved, *how* to make them productive and efficient, and *how* decisions are made, and *how* tasks are accomplished. Qualitative support does not undervalue the importance of the product, since the product becomes the focus around which these essentially human design and development processes will be organized. Improving software quality involves understanding the human processes underlying software development (cf., Krasner and McInroy 1989; Curtis, Krasner, and Iscoe 1988; Krasner, et al. 1987), providing methods and technologies to support these processes, and managing them effectively. This is what we mean by *process management*. Producing a well-designed deliverable becomes the by product of a well-managed set of human processes supported properly and organized around focused objectives.

Large-scale software development projects are usually managed to three principle factors: cost, schedule, and technical performance. Although all three are highly interrelated, in this paper addresses technical performance in isolation. Technical performance is, in turn, concerned with (1) quality of the product and (2) productivity of the people. Figure 1 presents this breakdown. The goal of our research is to develop process management knowledge and recommend process models that can yield improvement both in the productivity of developing software systems and in the quality of the end product. Unfortunately, neither productivity nor quality have yet been well defined for most software process research. Quality may be the most vague and elusive concept; however, everybody seems to think that quality is very important and that they know it when they see it. Advocates of Demming (1982) assert that quality leads productivity, and, therefore, we present our characterization of software quality first.

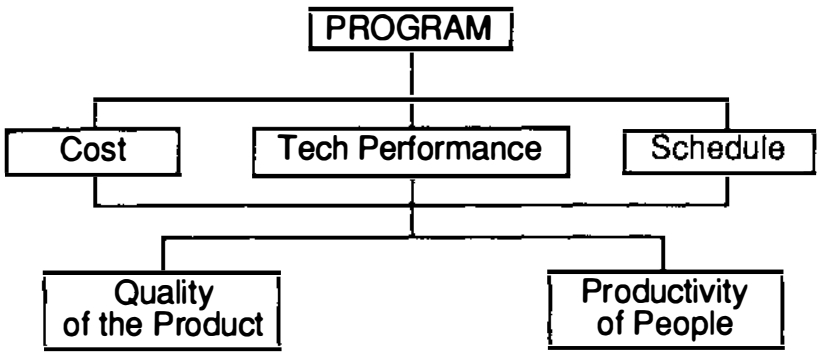


Figure 1. Program Performance Criteria

SOFTWARE QUALITY

The notion most often related to the value of a software system is its *quality*. Crosby (1979) defined quality (in general) as conformance to requirements. Although this may be a consistent definition, it is not of much practical use since it does not address what the requirements ought to be (i.e., the scope and quality of the requirements themselves). A definition more useful for our purposes here may be: *quality is the degree to which a system conforms to a set of predetermined standards (i.e., requirements, constraints,etc.) relating to the characteristics that determine its value and its performance for the function for which it was designed.*

In other efforts, computer scientists have been involved in characterizations of software quality that can serve as the basis for design metrics to provide guidance and prediction of product attributes. These characterizations attempt to decompose quality into its constituent parts so that the concepts can be better understood and new metrics defined. The two best known such models are Boehm (1978) and McCall (1977). The McCall framework continues to evolve under the sponsorship of RADC and is related to the U.S. Air Force Quality Indicators effort. The latest version, along with the set of related research issues, may be found in Pierce (1988). From a statistical control point of view, quality is concerned with measurable aspects such as errors, faults, and defects (and their associated rates). In that sense, a number of software practices have been shown to be correlated with these measurable properties thereby giving us the strongest evidence of good and bad affects. We are aware of published summarizations of the relationship of defects to other system attributes. For example, Jones (1986) indicates the relationship of defect rates to system size. The disappointing fact, however, with all of these statistically based efforts is that the notions of defect (and error, fault) are still not sufficiently or consistently well defined to have significantly penetrated the large community of software practitioners. Even though the notions of software quality are still ill-defined, we will refer to the factors introduced in the following paragraph to provide substantiation of our results.

Software quality evaluation is currently highly subjective and judgmental, although some quantitative measurement approaches are being tried. Measurement technology is far from practical usefulness. The evaluation of design solutions is commonly done in software engineering courses, programming contests, and product design reviews. Although such evaluations are subjective, there exist commonly used criteria for judging a system's correctness and quality. The following are major factors in our decomposition:

(1) The software works according to specified requirements (i.e., it is fast, efficient, and functional as required).
(2) It is evolvable with respect to lifecycle factors (i.e., adaptable to change, usable, etc.).

(3) The design process is well formed (i.e., esthetics) and the related information (e.g., documentation) ensures that factors (1) and (2) are achieved.

The primary criterion within factor (1) is correctness in terms of meeting the stated (or inferred) requirements with a feasible product design and implementation. Evolvability criteria deal with the system from three points of view, namely, the system in operation (reliability, efficiency, security, usability, and robustness), the system in transition (portability, reusability, and interoperability), and the system in revision (maintainability, flexibility, and testability). The relative importance of these criteria is typically derived from the stated or inferred requirements about the system's life-cycle. Information esthetics describes the state of the product's associated materials. Are the associated documents compatible? Is the delivery complete in terms of all the requirements? . . . documentation, training aids, storage containers, etc.? This decomposition of quality is diagramed in figure 2.

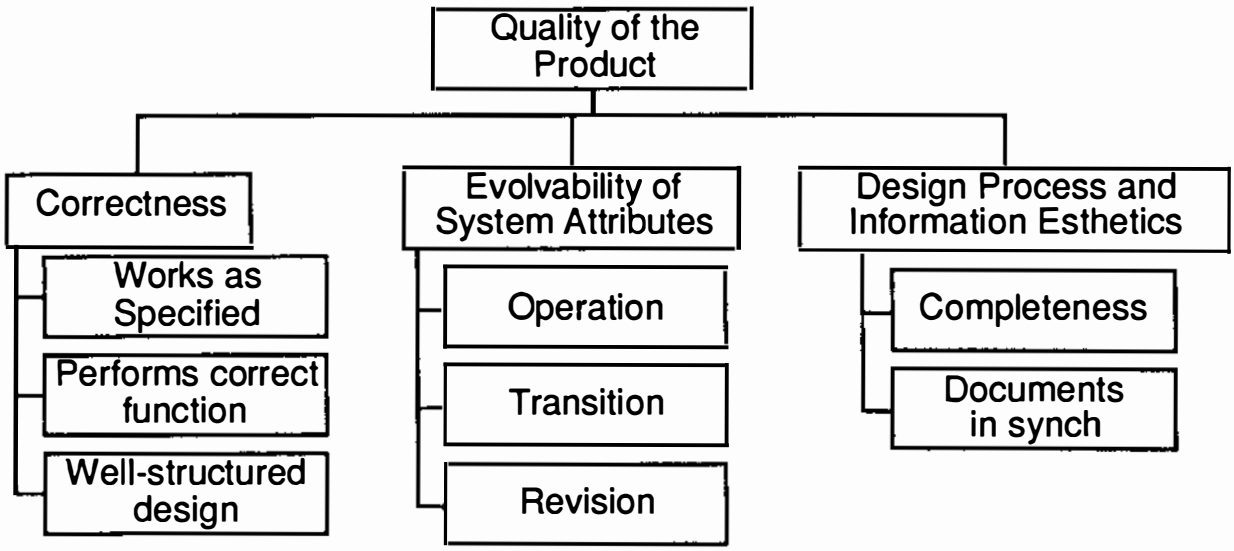


Figure 2. Decomposition of Quality

PRODUCTIVITY

Productivity is a more popular subject, both in reported research and in number of suggested solutions. Nevertheless, there is not common agreement of what productivity is. The classical definition of productivity–*output (value) divided by input (cost)* –is properly applied to manufacturing, but it fails to adequately characterize software engineering activities. Four other commonly used definitions of productivity are:

1. A measure of output per unit of labor (e.g., lines of code/man-month)
2. Dollar value per unit of output (e.g., $/module)
3. Output per calendar time (e.g., products/year) and
4. Calendar time per output (e.g., time-to-market/product).

Many attempts have been made to identify–and sometimes measure–the factors that impact the productivity of software development (e.g., Boehm 1981). Such studies appear to be based on an approach where the objective was to measure the development productivity (cost) impact of various elements of the development process or environment, rather than the objective of understanding and addressing the fundamentals of software development (see Walston and Felix

1977). In other words, a lot of measurements have been made, but no theory has been established. It is not clear whether any model–or hypothesis–of productivity was established that could be tested against the measurements. In these studies, productivity is tied to specific ways of development without any clue as to how productivity might relate to different ways of developing software. The areas, factors, and issues identified by our research are somewhat different in that they focus specifically on the aspects of software development where the greatest potential for productivity improvement is believed to be found. However, few measurements or analysis have been made to quantify the potential improvements, and the question of what aspects of productivity particular solutions address is still not answered.

A first attempt at a top-down analysis of productivity is presented in figure 3. The three major factors determining the development time (productivity) are (1) the amount of work to be done (as represented by the deliverables, as well as the process of getting there), (2) the amount of rework (waste) due to mistakes and changes, and (3) amplification of the capabilities (effectiveness and efficiency) of the developers in their environments. It is not clear whether these dimensions represent a complete taxonomy, however they are certainly related. These major components can be further decomposed. The factors that amplify the designers' capabilities are divided into two categories: (1) knowing what to do and how to do it and (2) doing it faster and better. The first category concerns the knowledge and skills that can be applied to the problem and solution domains, as well as the method or process of design. Without this expertise the problem cannot be recognized or solved. The second category deals with efficiency: How quickly can the designer find the required information? How easy is it to comprehend? and How quickly can the information be changed, synthesized, or analyzed to provide the desired result? Similarly, the components of reducing rework and the amount of work needed in the first place are also decomposed in figure 3.

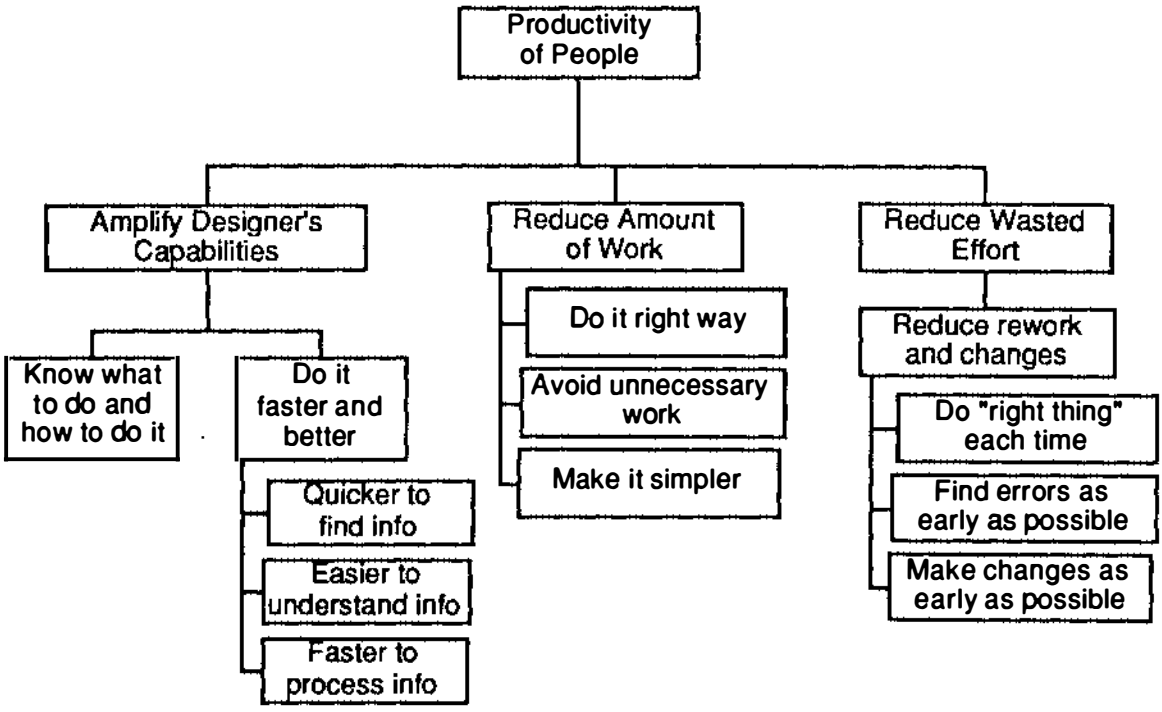


Figure 3. Decomposition of Productivity

PROJECT PROCESS MANAGEMENT

A project is usually described in terms of the organizational functions that make them up (e.g., project management, finance, quality assurance, logistics, test, engineering–within engineering there is systems, hardware, software, etc.) However, commensurate with current innovative thinking, we propose to view projects in terms of their processes. In some cases, it may be the same thing (e.g., configuration management). In other cases, it may appear to have no correspondence at all–like viewing a deck of cards as being made up of 13 ranks, rather than 4 suits (e.g., cost and schedule management are processes that cross all the functional elements such as engineering, test, etc.). In doing this, we shift our management focus from monitoring each of the respective functional organizations to monitoring the required processes that must be taking place; hence, we arrive at the name *process management*. Some of these processes are well known and are currently being well managed, they are what we call *conventional management processes* (e.g., configuration management). However, we find that the conventional management processes do not adequately address all of the quality and productivity factors described in the previous section. **While there are numerous processes going on in projects, most of which are not controlled, it is our thesis that certain ones need to be overtly managed to ensure specific technical performance.**

We have empirically identified below the key processes of technical performance that will need to be explicitly defined, monitored and controlled within our new paradigm for process management. These fall into three general categories of management: roadmap, coordination, and design. Roadmap management deals with: objectives/plans, exploration, risk, and change. Coordination management deals with: customer/user interactions, internal task coordination, and coordination with external organizations. Management of design includes decisions (e.g., tradeoffs, issues, rationale), methods and notations, tool acquisition and use, and abstraction/structures. Figure 4 shows these processes. The broader notion of managing a set of software projects as a coherent business, including the management of the existing and potential software asset base, is also shown.

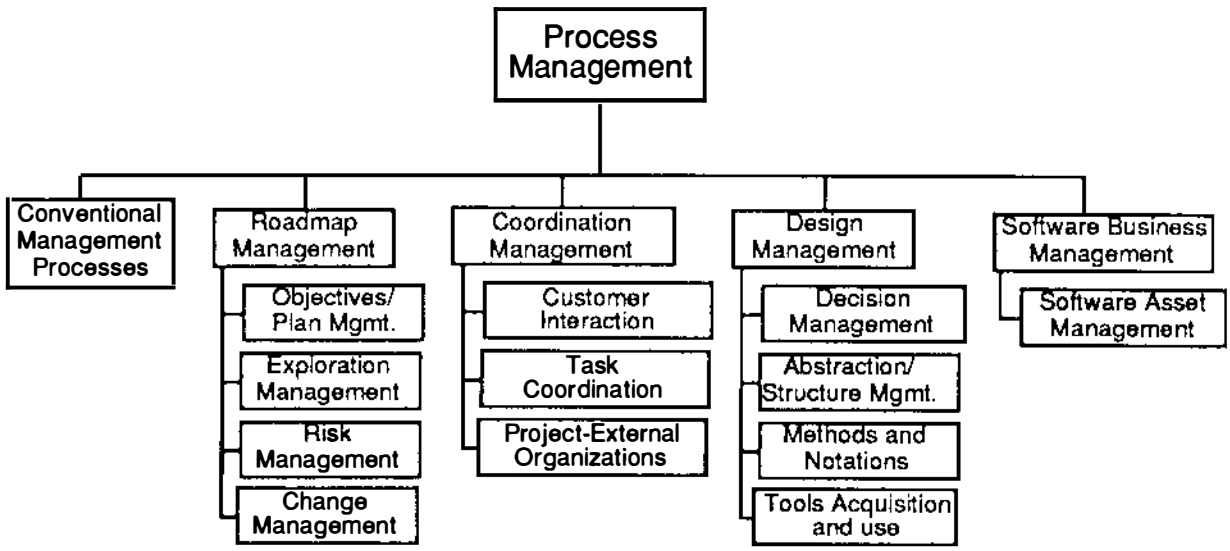


Figure 4: Process Management

In order to completely describe this approach, we must describe how these processes satisfy quality and productivity goals, how the processes might be implemented, how progress or status is measured, and finally, how associated measurements are analyzed and fed back into the

management process, as is graphically demonstrated in figure 5. Doing this for all the quality and productivity factors and all the processes is beyond the scope of this presentation. We have therefore selected one process to develop here, leaving the rest for future description.

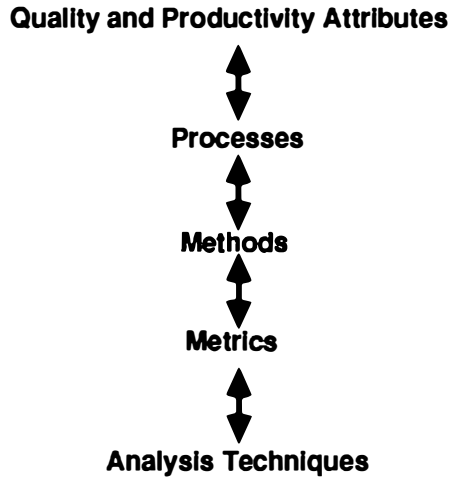


Figure 5. Process Development

Roadmap management processes have been addressed initially in work such as Boehm's (1988) risk management. We see that these developments will continue to evolve into a more complete approach . Therefore, we will not expand this area, but rather we select an area in more need of initial definition. The need for coordination management, the second group of processes, has been recognized informally, and most managers attend to these in their personal management style through staff meetings, staff functions (assignments), etc. With the current increasing recognition of its importance, coordination management will become more institutionalized and a formal part of the process. As informal coordination processes are identified and described, they will become standard operating procedures and eventually be codified in *groupware* (Krasner 1986; Krasner and McInroy 1989). Management of technical design, however, is the least recognized and least developed area of process management; hence, it will be the focus of this section. We have previously observed that one of the key processes ripe for improvement is design decision management, the process by which technical design decisions are:explored, made, unmade, etc. (Curtis, Krasner, Shen, and Iscoe 1987). This is the focus of the remainder of this section.

Software Design Decision Management

In a recent review of the psychological aspects of individual decision making under uncertainty (Hink and Woods 1987), it was shown that individual performance is usually suboptimal in both novices and experts. Specific problems are exhibited in suboptimal choice strategies, miscalibrations in probability assessments, incorrect statistical inference, and inconsistencies in preferences for uncertain outcomes. Furthermore, in situations that require team and large project performance (i.e., most industrial software efforts), these effects are significantly amplified as misinterpretations in many other tasks. In higher level organizational models of decision making, it is assumed that a rational process is used where the confidence in a particular decision can be bounded by quantitative estimates of the uncertainty of the information it is based on and the stability of the environment surrounding it. The decision-making models presented in the literature (e.g., consensus management, steam-rolling, political factions) only partially describe the process of reaching design decisions on large, complex software system projects. In previous studies (e.g., Curtis, Krasner, and Iscoe 1988), we found the assumptions behind current models of business-oriented decision making to be false in the domain of software engineering .

Problems in Software Design Decision Making

In those studies we identified a number of aspects of large software system developments that contributed to design decision making difficulties; these were due to project dynamics, complexity, affinity towards non-decision making, and team conflict. Although these four areas are closely related, we present each as separate for convenience.

Project dynamics induced difficulties. It was observed that change from sources external to a large project appeared to be extremely disruptive because it created greater uncertainty and lack of knowledge about the operational context that bound the requirements and design choices. The stability and quality of technical decisions and plans were impacted by these changes. The following factors have been seen to complicate technical design decision making: (1) the instability of the environment in which the decisions will be implemented and the fluctuation in requirements this causes, (2) the level of uncertainty surrounding the implications of early decisions and the length of time before feedback is received, (3) the complexity of the interactions among decisions, (4) the intertwining of technical and organizational objectives, (5) the need to invent as opposed to the possible availability of existing solutions, (6) the extremely rapid shift in the underlying technology base, and (7) developers discarding information crucial for later technical decisions because they do not understand its relevance. Change-stimulated difficulties in design decision making seem to cause protracted iteration and rework later in development. Because the quality of early design decisions are a crucial factor in the amount of rework, extra effort was required in later stages of development.

Complexity induced difficulties. Making stable technical decisions on large projects was observed to be difficult due not only to externally induced change, but also to the complexity of the cost, schedule, functional, and performance constraints that needed to be mutually satisfied. The uncertainties caused by missing, conflicting, or fuzzy information about requirements and possible changes confounded trade-off decisions on large projects. The most difficult technical design issues addressed in studied projects were those that involved trade-offs among the following constraints: (1) time and/or space (e.g., real-time behavior, limited space for code/data, response time), (2) environment interaction (e.g., interaction with the physical world, hardware-software interface, human operators), (3) quality (e.g., level of fault tolerance, design flexibility), and (4) algorithmic performance. The nontechnical constraints placed on top of technical issues complicated and altered the rationale for making various decisions. The following typically were often cited: (1) market constraints (e.g., maximum cost to customer, market window, installed equipment), (2) organizational constraints (e.g., synchronization with, or dependencies on, other developments, strategic marketing plans, critical staff availability), (3) externally imposed constraints (e.g., regulations, development standards), and (4) productivity concerns. Although better tools were needed to support the analysis required for many technical decisions (product performance being a primary issue), it was the complications added by these nontechnical constraints that generated the most complaints in our studies. Most professionals found it difficult to perform effective trade-off analyses due to a lack of rigorous (and stable) criteria about nontechnical issues. This led to the predominantly seat-of-the-pants methods where collective experience became the key determinant.

Non-decision making. Much non-decision making was observed. Postponing or not making decisions because of uncertainty in earlier stages resulted in the limiting of implementation options in later phases. An example of this is seen in the following statement made by a software section manager: *"One of the pitfalls in our process occurs when we (e.g., marketing, engineering, development) say, Do we have to make the decision on how it's going to operate? Could you (software team) write it both ways? We say, well it's going to cost some resources, but we could. The tendency is to not make the decision because we can have it operate both ways. This leads to thinking that we can make everything flexible. In implementation we can do fewer things because we are doing each thing eight different ways in order to be flexible."*

Currently, various forms of prototyping are used to obtain some performance and resource usage answers. However, prototypes appear most valuable for the human interface where mock-ups are common. The closer the prototype is to the functionality of the final system, the more expensive it is to build and the harder it is to throw away, even when requirements feedback indicates a large redesign effort. In several cases, we observed that a hacked-together prototype became an unwieldy product because the decision to redesign was not made. Frequently, prototypes were developed because information about the operational usage of the application system was not available or had not been communicated to the development team by those who had interacted with the customer. Prototyping, as a technique for better defining requirements, may work in situations where the requirements are not understood by the customer or user organization.

Conflict based difficulties. The key role of conflict in the design process is just becoming understood (Walz 1988). Problems sometimes arise from conflicting assumptions about requirements in multiple team situations. In some cases, projects have been significantly set back when simulations of the same system by different organizations gave radically different sets of results. The resolution of the different assumptions that were built into these models took months.

The wrong choice of upstream design representation can be disastrous downstream. Many senior systems engineers emphasize their aversion to legislated representational formats until they have decided which phenomena were the most crucial to represent in the design. Having made this determination, they select a format that facilitated the representation of these issues; however, shifting priorities due to change make this selection difficult to finalize. A correct long-term choice enables technical coordination via a shared model of the product structure or behavior.

A Design Decision-oriented Process Model

In order to manage a process we assert that it is necessary to have a defined model of that process and subsequently to be able to evaluate the actual process against the model through measurements. The current generation of large-scale software process models do not have the latter capability nor or they driven by quality and productivity concerns (e.g., Royce 1987).

Early design decisions are a crucial factor in the amount of rework required in later stages of development. Focusing on decision making generates a much more atomic model of the software process (decisions versus large-grained development steps) than is implied in the general literature. Most models of the design process imply that design decisions should be made hierarchically in a top-down, breadth-first or depth-first fashion. Our studies at both the team and individual levels suggest that this is typically not the way designers work. They may try to rationalize their efforts top-down, but the pattern of making their decisions is much more opportunistic and scattered as targets of opportunity for closing decisions that had been tentatively posed. Figure 6 shows a model of a single decision as commitment to some additional elaboration of the artifacts associated with the development.

In this model, decisions are further bounded by hard constraints, negotiated constraints, and dependent (and variable-strength) links to other decisions. Decisions therefore cause not only artifact elaboration but the generation of other dependencies. Thus, a network of tentative decisions is generated whose resolution depends on stabilizing the dependencies that act as linking relations among the various alternatives of the decisions comprising the network. This is viewed as a constraint manipulation process over the net. Instead of formal reasoning about decisions the process tends to be one of developing a web of dependencies among decision alternatives and selecting the set of alternatives that minimize the dependency conflicts.

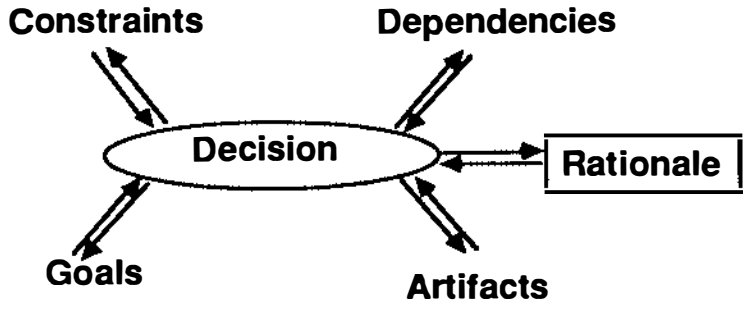


Figure 6. The Elements of a Technical Decision

Decisions are bounded by dependencies, and their uncertainty is reduced by overlaps in dependencies. Thus, sets of decisions are made together rather than in a strict sequence.

When the optimal choice for a decision is not obvious, several alternatives with associated rationale may be posed. (If designers cannot pose alternatives, they may not be aware that they have made a decision.) These alternatives may be resolved through simulation or some other form of analysis. Frequently their resolution depends on how the dependencies created by the alternatives of this decision interact with dependencies created by the alternatives of other decisions. Figure 7 presents such a decision network, where the links represent the dependencies between decisions. The relationships between decisions and goals are not shown.

The problem of stabilizing the design lies in deciding on a feasible set of alternatives for satisfying the web of dependencies while not violating the constraints. In many areas of engineering, automated techniques exist for resolving such dependency networks (e.g., linear programming). Unfortunately, similar techniques have not been developed in software engineering.

Once instantiated in code, these dependencies become the constraints that bind the redesign of the system during later development activities for correcting or enhancing. Thus, activities that were conducted informally, possibly by a single individual during design, set boundaries for later steps whose process is much more formally controlled. The process for an enhancement may be more difficult than that for a correction because it typically represents a change of a greater number of design decisions and, thus, a readjustment of more of the dependency network.

The relationship of this approach to currently emerging risk management techniques can be stated as follows: The identified risk items are mapped into the issue oriented rationale basis for the decisions to be made. Therefore, they also become a part of the inherited dependency information between decisions. We believe that risk management depends upon design decision management.

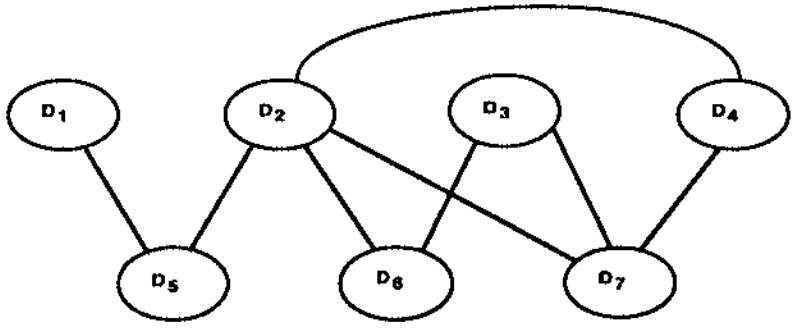


Figure 7. The Decision Net (Actual links are between alternatives.)

Decision Examples

The management of decisions becomes more formal when decisions can be represented in a processable form. It is informal while they are stated in natural language and communicated verbally among a small team of individuals. Decisions are managed more formally (through configuration management) as they get embedded in running code. Finally, the people to whom decisions are communicated change over time from a small team of system designers to a project design team, to a programming team, and finally to parties affected by a change in the operational system (i.e., users and maintainers). The problems of decision making with respect to change are clearly complicated by an organization's internal information flow problems.

Consider some of the following major types of decisions that are made (and remade) at various points in the software development process: (1) modeling the requirements and functional specification, (2) representing and specifying the design, (3) deciding on the disposition of prototype code, (4) choosing an implementation technology for the design, and (5) prioritizing the corrections or enhancements to be included in the system.

The management, control, and communication of decisions during the performance of each of these stages differs. The management of decisions becomes more formal when decisions can be represented in a processable form.

The number of people involved in the communication net for a change decision can differ dramatically between small and large projects. Thus, the control and communications mechanisms required to support the social processes underlying the development process differ by project size. These mechanisms must become more formal and robust as the size of the project increases. This formality is driven by the need to provide a stable base of design decisions from which to continue making decisions that elaborate the artifacts. Failure to provide this stable base will compound the rework effort because new decisions will be based on decisions that, unbeknownst to the decision-maker, have been changed elsewhere. The process of development becomes an effort to localize (and thereby control) the iteration of decisions to ever smaller portions of the system.

One problematic of our approach is captured in the following questions: Which decisions out of potentially thousands made on a large project should be explicitly managed? What constitutes a committed decision rather than a widely (or personally) held assumption? And when does an assumption become a defacto decision? Such questions are problematic, however in the context of our team approach, a practical guideline must be that a decision to be managed is one that needs to be analyzed by other experts or is one in which the resultant constraints or dependencies affect other members of the team or other work groups.

While most process models describe software development as a set of phases that is terminated by the existence of a set of artifacts (committed decisions represented in documentation, code, etc.), we wish to model the software process by looking at how decisions are reached while developing the artifacts. The design process that emerges from this model is one of developing a web of dependencies among decision alternatives and selecting the set of alternatives that minimize the dependency conflicts. Decisions are bounded by dependencies, etc., and their uncertainty is reduced by overlaps in dependencies. Thus, sets of decisions are made together and in a parallel, yet coordinated, fashion rather than in a strict sequence.

The Design Decision Support Approach and Software Quality and Productivity

We have argued that quality and productivity factors, such as correctness and evolvability, can be dealt with directly through a process management approach to design decision making. We have

also described the team-oriented method for implementing that approach by the use of several examples. We assert that such processes and methods may subsequently be evaluated (i.e., measured) by the use of subjective process indicators such as the stability of the decision network, the number of tentative or volatile commitments, and the rate of unresolved issues. The techniques for collecting and analyzing such measures are currently the subject of one of our research programs (cf., Pore and Krasner 1990, to appear).

By opening the key decision-making process to expert teams, we expect to get higher quality decisions that lead to higher quality systems (Shaw 1981). It has been shown in the development of information systems that teams supported by group decision support technology produce decisions that are of better-than-average quality (DeSanctis, et al. 1988). We have previously shown that in large, complex software system developments diverse knowledge needs necessitate a group interaction and negotiation approach (Krasner, Curtis, and Iscoe 1987). In such situations, *system superconceptualizers* are few; therefore, it is expected that the negotiated design decisions of the expert team will, with very few exceptions, be the best approach. This is directly seen in the number of architectures considered and the hybrid model that usually emerges.

There is currently high interest within the software development community in the development of meaningful metrics that provide a means of management awareness, program control, early maintenance predictability, compliance to reliability requirements, and other project management issues. The need is for a set of metrics and associated analysis techniques that adequately describe all the properties of interest about the software system. This has led to a proliferation of proposals, workshops, conferences, and publications about software metrics. However, it has not (thus far) led to the development of any acceptable and widely used new measures.

One example of a metric development running amuck is the notion of software reliability. This metric was derived from its hardware counterpart to reflect the quality of the final product. Software development managers are quick to point out the modeling problems (hardware waring out creates faults, software doesn't generate faults with time) and the reasons that reliability (probability of a faultless execution) does not address their management concerns. While software reliability requirements are sometimes levied on new contracts, the measure is evidently without sufficient merit as a management tool either for development or for estimating maintenance costs. It has been required in the past simply due to a lack of alternatives. What is needed to justify the use of reliability is an analysis technique that the software development manager will accept. Without it, the metric is misleading and inappropriate.

As an alternative to reliability, we propose a metric that describes more accurately what the customer has in mind when levying reliability requirements. We are engaged in a research effort to develop an "expected cost to maintain" metric. This measure can be implemented to account for the complexity of error types, their varying frequencies, and their respective costs to fix. In this respect, it is more refined then reliability (uses more detailed information, and models the real world more accurately), and it brings to bear an entirely new type of information (degree of fault damage).

Both reliability and expected costs are derived from fault counting (a more elementary metric); hence, they represent different methods of analyzing the fault count data (or early in the development it would be projections of expected faults). This illustrates the principle that the value of a metric (e.g., fault counting) is not intrinsic but determined by the analysis technique that is applied to it. Reliability analysis does not consider distinctions in fault types and damage to repair; whereas, expected costs weights these to balance risk and possible return.

The measurement of the quality of a software system can now be modeled accurately and projected (early in the project) with a metric that can then be used to compare alternatives in the design decision making process.

CONCLUSIONS

Our new notions of process management affect the type of information needed in a computer-based support environment. The information requirements include the storage of several new types of informations, these include: responsibilities, commitments, skills/expertise, issues (open, resolved, and dissolved), decisions, assumptions, goals and objectives, and shared models of product and process. These will require the development of new technology to handle the different types of data (e.g., voice, video, and pictorial) in an integrated fashion. This capability does not exist currently. In the future, such environments will be driven by embedded process models.

Specifically for the support of design decision management, tools are needed to help represent the complexity of these dependencies and methods to reduce the breadth of their effects across the decision network. These tools should reduce the amount of rework necessary during design and more clearly present the dependency structure to implementors and maintainers. In order to automate some of this dependency resolution, the semantics of these dependencies must be represented in processable form. This is the goal of many knowledge-based approaches to software development (Bimson and Burris 1989), but the approaches are far from providing such tools for designing large systems. These dependencies are not always visible from design documents or code. We may reduce subsequent rework by increasing the visibility of dependency violations among the existing network of design decision wrought by changes to the code. If we can model the process of revisiting design decisions, we may be able to predict the difficulty of rework during later development steps.

Human decision making under uncertainty has been shown to introduce bias, distortion, and error. It has been suggested (Cohen 1985) that AI techniques can be used to make decision making more objective, and expert system approaches are currently being explored in other, more narrow domains. However, the domain of complex software design presents many formidable challenges to this technology (Mostow 1985; Barstow 1987) and is not expected to contribute significantly to solving the problems described here. See Bimson (1989) for an interesting counter example.

Change and uncertainty problems that are due to new, missing, conflicting, or fuzzy information regarding both the customer's needs and environment result in design decision problems. This leads to further problems managing the mapping of this information into an evolving design and subsequently the organization's communication processes. Normal project practice is to encapsulate deigns decisions and associated potential changes within specific components (Parnas 1985) and encapsulate uncertainty by making the associated system components table-driven or customizable by the end user. In addition we support an approach that makes the decision space a team-shared information base.

It has been proposed that issue identification and handling might be the basis of the next generation of design decision support systems (McCall 1986). Identifying and capturing the many complex issues that are left afloat during design would be a first step toward machine-aided reasoning on design management. Handling technical conflict within a team environment can also benefit from issue capture and analysis. An issue-based decision support environment needs to be supplemented with tools that allow interactive explanation, negotiation, assumption surfacing, and closure on the decisions in user terms rather than in development terms.

A description of technical performance was given in terms of the described processes that affect quality and productivity. The association of specific quality and productivity attributes with corresponding processes directs development projects to specific activities in implementing corrective measures and, hence, to a project management tool. Further research is being performed to (1) develop metrics for the described processes and their related factors and (2)

develop process management tools to facilitate the explicit management of the recommended processes.

Previous studies and experiments in the open literature have not generated the information necessary to analyze processes on extremely large projects. We assert that data gathered from actual projects is needed for assessing the impact of different factors on development process and project outcomes. The continued observations from our empirical studies effort are driving our research program of exploring, synthesizing, and supporting process management approaches and models of software processes.

REFERENCES

Barstow, D. 1987. "Artificial intelligence and software engineering." In *Proceedings of the 9th International Conference on Software Engineering,* 200–211. Washington: IEEE Computer Society Press.

Bimson, Kent D., and Linda B. Burris. 1989. Assisting Managers in Project Definition: Foundations for Intelligent Decision Support. *IEEE Expert* 4(2): 66-76.

Boehm, Barry W., et al. 1978. *Characteristics of Software Quality.* Amsterdam, The Netherlands: North-Holland Publishing Co.

Boehm, Barry. 1988. A Spiral Model of Software Development and Enhancement. *Computer* 21 (5): 61-72.

Burrill, C. and L. Ellsworth. 1980. *Modern Project Management: Foundations for Quality and Productivity.* Tenafly, N.J.: Burrill-Ellsworth Assoc.

Cohen, P. 1985. *Heuristic reasoning about uncertainty: An AI approach.*, Boston: Pittman.

Crosby, Philip B. 1979. *Quality Is Free.* New York: McGraw-Hill Book Co..

Curtis, Bill, Herb Krasner, Vincent Shen, and Neil Iscoe. 1987. "On Building Software Process Models Under the Lamppost." In *Proceedings of the 9th International Conference on Software Engineering,* 96-103. Washington, DC: IEEE Computer Society.

Curtis, B., H. Krasner, and N. Iscoe. 1988. A field study of the software design process for large system. *Communications of the ACM* 31(11): 1268–1287.

DeSanctis, G., G. Dickson, and R. Gallupe. 1988. Computer-based Support for Group Problem-finding: An Experimental Investigation. *MIS Quarterly* (June): 277-296.

Deming, W. Edward. 1982. *Quality, Productivity, and Competitive Position.* Cambridge, MA: Massachusetts Institute of Technology, Center for Advanced Engineering Study.

Hink, R., and D. Woods. 1987. How humans process uncertain knowledge: An introduction for knowledge engineers. *AI Magazine* 8 (3): 41–53.

Jones, C. 1986. *Programming Productivity.* New York: McGraw-Hill Book Co.

Krasner, H., Shen, V., Iscoe, N. and Curtis, B. 1986. "Preliminary Observations from the MCC Field Study of Large Software Projects," *MCC TR# 390-86(p),* Dec. Microelectronics and Computer Technology Corporation, Software Technology Program.

Krasner, H., B. Curtis, and N. Iscoe. 1987. "Communications breakdowns and boundary spanning activities on large programming projects." In *Proceedings of the Second Workshop on Empirical Studies of Programmers,* 47–64. Norwood, New Jersey: Ablex Publishing, 1987.

Krasner, H. B. 1989. "Requirements Dynamics in Large Software Projects: A Perspective on New Directions in the Software Engineering Process." *11th World Computer Congress (IFIP89),* August. Amsterdam, The Netherlands: Elsevier Science Publishers, B.V.

Krasner, H., and J. McInroy. 1989. "Groupware for Software Process Management: Research and Technology Issues," to appear in *Proceedings of the IFIP Workshop on Groupware Technology,* Aug. 1989, San Francisco, CA: International Federation for Information Processing.

McCall. 1977. *Factors in Software Quality.* GE-TIS-77CIS02, General Electric Co.

McCall, R. J. 1986. Issue-serve systems: A descriptive theory of design. *Design Methods and Theories,* 20 (3): 443–458.

Metzger, P. 1973. *Managing a Programming Project.* Englewood Cliffs, N.J.: Prentice-Hall.

Mostow, J., ed. 1985. Special issue on AI and software engineering. *IEEE Transactions on Software Engineering,* 11 (11).

Parnas, D. L. 1985. *Software Aspects of Strategic Defense Systems,* Department of Computer Science TR# DCS-47-IR. (July) Victoria, B.C., Canada: University of Victoria.

Pierce, P. A. 1988. "Software Quality Framework Issues." Documents from the RADC task force on Software Quality Issues, Vol I,II,III (October). San Diego, CA: SAIC.

Pore, M. and Krasner, H. 1990. The Decision Theoretic Approach to the Analysis of Software Development Metrics, submitted to IEEE Software special issue, March , 1990

Royce, W. W. 1987. "Managing the Development of Large Software Systems." In *Proceedings of the 9th International Conference on Software Engineering,* 328-338. Washington, D.C.: Computer Society Press of the IEEE.

Shaw, M. E. 1981. *Group Dynamics.* 3rd ed. New York: McGraw-Hill Book Company.

Walston, C.E., and C.P. Felix. 1977. A method of programming measurement and estimation. *IBM Systems Journal* 16(1): 54-73.

Walz, D. 1988. A Longitudinal Study of Group Design of Computer Systems. Ph.D. diss., University of Texas.

IMPLEMENTING ARCHITECTURAL METHOD TO DELIVER HIGH QUALITY INFORMATION SYSTEMS

Robert E. Shelton
CASEware, inc.
Post Office Box 8669
Portland, Oregon 97207
(503) 626-6326

ABSTRACT:

This paper presents a practical approach to developing and maintaining high-quality information systems based on construction-industry architecture. Architecture provides a framework into which established systems methods are integrated, and through which systems engineers can improve the quality of the process by which Information Systems are produced.

BIOGRAPHICAL SKETCH:

Robert Shelton is Managing Director and co-founder of CASEware, inc., architects, integrators, builders of systems. CASEware specializes in the rapid engineering of high quality information systems using systems engineering method and Computer Aided Systems Engineering tools. CASEware provides extensive support to organizations implementing systems engineering and CASE.

Shelton's background is both technical and managerial, including five years as a project manager and systems engineer with Pacific Bell. He has worked as a systems engineer in a range of industries, including computer manufacturing, academic research and the defense sector. Shelton has also operated two private businesses and actively participated in the formation and operation of three corporations.

Shelton received his Bachelor of Arts degree in Psychology from Stanford University, specializing in neuropsychology and computer science. He is a member of IEEE and ASM.

INTRODUCTION

This paper presents a practical approach to developing and maintaining high-quality information systems based on the technology of construction-industry architecture.

Architecture provides the framework into which techniques from established systems engineering methods integrate. Many of these techniques are already familiar to a significant component of the Information Systems community. Architecture itself forms a missing link connecting these tools: a basis for understanding the process of systems engineering and the reasons for using particular engineering tools. Architecture is an effective structure for explaining systems engineering, and for guiding the actual implementation of that work.

Architecture also becomes the basis for Computer Aided Systems Engineering (CASE) tool selection and implementation. CASE is automation of the process of systems engineering. Architecture itself is a method for engineering complex systems. A proven and understood engineering process is a prerequisite to automation of that process: we cannot successfully automate process we do not understand.

This paper addresses implementation of Architectural method in the systems environment, and explores some of the challenges and issues to be resolved. This approach is drawn from experiences implementing Architectural method and CASE in diverse business and government environments.

TERMS AND CONCEPT

To facilitate communication, we should standardize our usage of three fundamental terms of this discussion: technique, method, and methodology.

Techniques are the most commonly applied systems engineering activities, and are frequently confused with method. Techniques are components of method: communication, diagraming, analysis and design tools. Examples of techniques are data flow diagrams (DFDs), Chen data models, Warnier-Orr diagrams, Bachman diagrams, structure charts and matrices.

Method is structure. Methods specific techniques to accomplish specific sub-tasks, as well as task-to-task relationships (transforms). Examples of methods include the Zachman Framework for Information Systems Architecture, Orr Data Structured Systems Development, and Structured Analysis/Structured Design.

To be precise, the Zachman Framework is actually a meta-method: a method which integrates methods (an thus numerous techniques). This Framework is fundamental to the Author's approach to Architecture.

Methodology, while abstractly defined as the "study of method", is more meaningful when understood as the systematic process for evaluating and improving method. In this sense, the Deming Management Method constitutes methodology.

Significant for our purposes in this paper is the Deming emphasis on the continuous commitment to process improvement. Implementing method alone is as insufficient as implementing technique alone. Technique in the absence of method is tooling without reason or structure. Structure (method) without improvement is stagnant, thus non-adaptive and non-competitive.

Our industry will experience long-term benefit on a meaningful scale only from the implementation of technique supported by method, itself supported by methodology. To wit, do it all and do it well, or don't expect significant improvements!

This paper will focus on Architectural method. Techniques will serve as the building blocks for method; Methodology is the quality control feedback mechanism.

THE OBJECTIVE OF METHOD

Method is structure. Method for systems engineering is the structure by which practitioners convert ideas into products. The products of systems are information views -- information used and shared to operate an enterprise. The product of our industry is information.

The need for structure arises from human creativity. We people are capable of tremendous variance. Humans usually don't perform a complex task the same way twice without structure. Structure empowers us to deliver consistent, repeatable, predictable results.

The objective of structure is quality. Developers of Information Systems must deliver high quality information reliably to stay in business.

Research (DeMarco, Deming, Jones, Zachman) suggests that the primary quality problem in engineering complex products is process failure -- the structure by which the product is delivered is measurably defective. As systems engineering is a human process, problems tend to be people process problems. Resolving the process problem should significantly reduce our industry's instance of product problems.

METHODS AND ARCHITECTURE

Systems methodologists have historically tried to reinvent engineering instead of leveraging the wealth of engineering experience available for the asking in similar professional fields. Because computers themselves have been perceived as special, unique and new technology, our profession has sought to develop new technologies for developing useful computer applications.

Systems methods are notable for what the DO NOT include. Each traditional (read this "commercial") method has addressed a segment of the Architecture process, largely to the exclusion of other components.

This practice has resulted in method "camps". Many in our profession have mistakenly inferred from the ensuing battle of method gurus that the camps of "process", "data" and "real time" are orthogonal axis in a three-dimensional world in which any given project (or worse, entire categories of business) must rest dominantly on one axis. This belief manifests itself in statements like "The utility business is dominantly data driven." Missing is the balance supplied by Architecture: integration of components required to achieve correct problem definition and correct solutions.

Where traditional systems methods are exclusive, Architecture is inclusive: notable for what it DOES include. An inclusive method provides the necessary and complete set of steps, models and instructions to convert ideas and needs into real working products. Architecture is structure around technique and transform.

Classical Architecture, as one might observe from the construction of a high-rise office complex (read this "system"), addresses the builder's equivalent of data, process, event and network in sufficient depth to deliver a correct product which performs as the buyer specified.

Inclusive approaches integrate more easily into existing environments (many practitioners already know part of the technology) and offer a more complete approach to addressing the engineering process than traditional exclusive methods.

Might such a thorough approach enable our profession to deliver more predictable products?

CLASSICAL ARCHITECTURE: AN OVERVIEW

Classical Architecture in the construction trades is a series of steps which organize and facilitate the transformation of concept to product.

In the construction of a building (and, as we will see later, in systems as well), three groups participate: owner/buyer, architect, and contractor/builder. These participants work together through the structure of the method of Architecture.

The steps of the Architectural process are reviewed here to familiarize the reader with the approach. Each step is evaluated for:

Purpose (Meaning)
Modeling Technology
Deliverables

Bubble Charts

The owner's initial concept and gross project scope are established in the first step, with the objective of determining that there exists a project to be discussed between owner and architect. Bubble charts depict the initiating request or need in an abstract, non-detailed picture in the owner's (NOT the technician's) terms. In this step, the Architect takes an idea expressed from the owner's mind to a "ballpark" view that can be shared and discussed.

Typical modeling technology is Bubble Charts, so called for the application of hand-sketched "bubbles" to depict rough space usage, relationships and building size. Deliverables are usually scratch-pad sketches, developed in a few hours or over a few days.

Architect's Drawings

The owner's initial concept and gross scope are refined in the second step to representations of the desired final product as envisioned by the owner. The objective of this effort is to reach agreement between owner and architect on the requirements for the final product. This step depicts the final product from in the owner's terms, and transitions sufficient knowledge to the architect to initiate design.

Modeling technology consists of drawings (pictures and artist's renderings), building cutaways, floor plans and scale models. Deliverables range from paper drawings and pictures to cardboard models representing the look and feel of the final building. Final product alternatives are evaluated through modeling.

Architect's Plans

Architect's plans are logical design for the building. The owner's view (how the building looks) is translated into a design representation (how the building will be constructed, of what materials, etc.).

At this step, modeling technology includes electrical and structural diagrams, site preparation maps, and a bill-of-materials for the building. The building design deliverable from this step is used for contractor (builder) cost, schedule and performance negotiations.

Contractor's Plans

Contractor's plans convert the building design to builder instructions -- a representation of the process required to construct the building. These plans are the building physical design and final proof of concept. Implementation (technology and physical) constraints are applied to the Architect's design to ensure a buildable product.

Blueprints are the common modeling technology. By following the blueprints exactly, the construction project manager and team can produce exactly the building envisioned by the owner.

Shop Plans

Shop plans are subcontractor out-of-context representations developed to support subassembly fabrication. Where a series of identical components are used in a building, these manufacturing specifications are the basis for "mass" production.

This step produces the patterns and jigs necessary to fabricate interchangeable parts.

Finished Product

Notable in a finished building are two factors the reader has likely observed:

construction proceeds in an orderly manner;

the completed building works -- failures are rare.

By application of Deming's work, the author infers this to result from a process (architecture) which itself works correctly most of the time. Failures can be identified, examined, and used as the basis for process improvement.

Architecture structures the process of converting an idea to a working finished product. The consistent results obtained from this process by the construction trades suggests a mature and relatively stable process. The author seeks to take advantage of the extensive (and expensive) learning that refined this process over several centuries, and proposes cross-application of the fundamental process of architecture to the development of information systems.

ARCHITECTURE APPLIED TO SYSTEMS

Engineering information systems is likewise reviewed step-by-step in light of the Architectural model. At each step we examine:

Purpose (Meaning)
Modeling Technology
Deliverables

Application of existing systems methods and techniques at each step based on the engineering requirements of that step greatly simplifies implementation and reduces the learning curve associated with new technology. So-called "brand name" methods are known and developed sufficiently that we can gain from the correct application of existing industry knowledge.

Bubble Charts

The owner's initial concept and gross scope are established, with the objective of determining that there exists a project to be discussed between owner and architect. Scoping can occur at the enterprise or application level in systems, with the effort at this level commensurate to scope. In either case, concept is the initiating idea from the buyer of the system or the owner of the business need. The objective is expression of the owner's concept in the owner's terms.

Modeling technology varies widely with practitioners. Textual approaches include Business Systems Planning from IBM. Diagrammatic approaches employ various representations of business data, function and event, such as Orr's entity and function diagrams, Flavin's Information Modeling, and high-level Object-Oriented and Structured Analysis. Deliverables model the business, need or problem under consideration; and to a lesser extent the current information systems.

In more information-sophisticated organizations, concept and scope derive from an enterprise architectural plan. The plan specifies those applications and information areas of most value to the business. Only applications in the plan are built (or bought). This Enterprise Information Architecture defines the classes of information and function most strategic to the owner's organization.

Architect's Drawings

The owner's initial concept and gross scope are refined in the second step to representations of the desired application as envisioned by the owner. The objective is to reach agreement between owner and architect on the requirements for the final product. This step depicts the final product from in the owner's terms, and is commonly called Requirements Definition.

Applicable modeling technology includes Information Modeling entity-relationship diagrams, Function Flow Diagrams, logical network models, state (event) models, and application scale models.

Application scale models are prototypes of three flavors:

paper drawings of screens, reports and menus;

non-working models on a computer;

working simulations on a computer.

These models all serve one objective: establishing a clear understanding between owner and architect of the desired final product. Note that the objective does NOT include coding of an application -- this use of modeling strictly excludes and strongly discourages what is known as "protocycling".

Deliverables from the Architect's drawings step can serve as the basis for package evaluation and selection, as well as the foundation for design and development.

Architect's Plans

Architect's plans are logical design for the application. The owner's view is translated into a design solution. In our industry, this design must address (at a minimum) data, function and network.

Modeling technology includes Chen Data Models, DeMarco Data Flow Diagrams, and logical network/data distribution diagrams. This logical design is used to evaluate physical implementation environments, application sizing and construction cost. This model should be used for cost, schedule and performance negotiation with design and construction firms.

Contractor's Plans

Contractor's plans convert the building design to programmer instructions. These plans represent the logical design as modified to reflect implementation reality. Such factors as operating system, data base management software, operations standards and practices, and hardware constraints must be considered to ensure a buildable product.

For these systems blueprints, common modeling technology includes Bachman data structure designs, program structure charts and network designs. By following the blueprints exactly, the project manager and programming team can produce exactly the requested application.

Shop Plans

Shop plans are out-of-context representations which support the use of reusable subassemblies. From the programmer's viewpoint, these subassemblies are library routines, window user interface development tools, and data base access routines. From the viewpoint of the owner and users of our systems, the entire application is a jig. Since our final product is information, not systems, an application is an information jig.

Finished Product

Most notable in finished information systems today is the absence of the two factors prevalent in finished buildings:

> construction proceeds in an orderly manner;
>
> the completed building works -- failures are rare.

By application of Deming's work, the author infers this to result from a defective process. Architecture is not currently applied in a rigorous manner. Improvement of the process would improve the product.

Architecture provides a structure into which the techniques and methods currently in commercial application can be applied. Missing today is the application of what we know. Our industry suffers not from the absence of method, simply the absence of systematic use of method.

IMPLEMENTATION: AN APPROACH

The fundamental steps required to implement architecture in an information systems organization are presented. These steps draw on experience implementing architecture, method and technique in information systems environments ranging in size from teams of two to organizations of several thousand.

Entrench Management Commitment

Active long-term commitment will be required from senior management, key systems managers and key user/client managers. Implementing architecture in an uncommitted environment usually produces failure.

One route to commitment is the application of architecture, method and technique on a small scale -- an "underground" effort on an existing project. Small successes can be aggregated over time to demonstrate the benefits and justify the costs of Architecture.

The lack of commitment is a human (not technical) issue. The lack of commitment may reflect a lack of information; often, an unspoken unwillingness to change underlies non-committal or resistive responses to Architectural method implementation.

Establish a Skunk Works

The Skunk Works team will lead the implementation of Architecture. This team of six or fewer must be composed of individuals willing and able to change themselves. Architecture, method and technique require the practitioner to follow specific processes and rules. People unwilling to change will not effectively adopt or implement these processes and rules. This team must be composed of the organization's very best. Excellent technical, social and team skills are required.

Train and Tool the Team

Tooling the team is crucial to success. Current Computer Aided Systems Engineering (CASE) tools are not perfect, but many do provide valuable support for diagramming, thinking, work organization and partitioning and communication. Tooling must be standardized within the team (and the organization). A variety of tools will be required -- no single CASE tool provides all required capabilities. Each team member must be supplied with their own tooling and workstation. Attempts to share primary tools prevent complete socialization and adoption of both the tool and the engineering practices which the tool supports.

The team will require extensive training together -- as a team -- in architecture concepts, systems engineering methods (NOTE THE PLURAL), and techniques. Training in each CASE tool to be used is essential -- unfamiliar tools will be used inadequately.

Expect, plan and budget for a learning curve of six months.

Select a Pilot Project

Select a project which will have meaning to the organization. Accomplishments on trivial projects will be ignored. Mission-critical systems of moderate complexity and scale are the best candidates.

The Skunk Works team must be dedicated full-time to this project. The project team must have a committed sponsor, and end users willing and able to participate actively. Do not select a project for which only uncommitted users are available. If the project does not have active participation from individuals the user organization "cannot live without", the resulting application will be one that the user organization CAN live without!

Reward Change

Most organizations attempt to reward performance; few reward change. The performance evaluation and reward system may need to change in order to implement architecture throughout an organization. At very least, the measures of performance must be specific to learning and successfully implementing architecture, method, technique and tools.

Apply Project Management

Effective metrics-based project management technology exists in other industries, thus will not be discussed here in detail. Suffice it to say that architecture as a process makes manifest the tasks of a project. With identifiable tasks, proven approaches to project management can be applied to information systems project management.

The project manager must be aware of two issues, and manage accordingly:

> Manage both the results and the process -- managing results alone will not produce process improvement;
>
> Measure process results for the purpose of process improvement -- develop and employ metrics to determine and remove defects in the team's implementation of Architecture, not simply the product.

CHALLENGES: WHAT PRICE QUALITY?

Experience implementing Architecture suggests that the most important change being made is improvement of the systems engineering process. While much literature today focuses on productivity improvement in the form of reduced development time, cost or defects, the linchpin is quality. Deming suggests that productivity and product quality result from process quality -- doing the right job the right way delivers the right product in the most timely and cost-effective manner.

Experience notwithstanding, significant challenges remain to be addressed by the implementor of Architecture.

CHANGE: The implementor is an agent of social change.

Systems projects fail for predominantly social, not technical, reasons. The same holds true for attempts to change the process of developing systems. Successful implementation of Architecture depends entirely on social change.

COST: This change is costly.

Training and tooling are required on a scale foreign to most organizations. The learning curve itself is costly. The "development" life cycle appears to take on "additional" work steps which themselves appear to add cost to each project. The choice to be made here is between the cost of doing the job right and the cost of continuing to produce poor quality products. In today's systems environment, continuing with present practices is costing numerous career systems professionals their jobs as organizations look for alternatives. Costly change is essential.

CASE: We must understand before we automate.

Automation of processes which are not understood produces incorrect automation solutions. Selection and implementation of tooling to automate method and techniques depends on first having a clear and practical understanding of the methods to be used. Attempts to use tooling to teach method will fail to teach method; the tooling will become costly shelfware. Method must be implemented before CASE.

RESULTS AND CONCLUSIONS

With the construction trades achieving in excess of a 98% success rate (measured by budget, time and product reliability), and some assessments placing the software industry's effective failure rate equally high, the financial and strategic impact of process improvement is significant.

Equally crucial are the lessons we systems engineers and managers can learn from the construction industry's attempts to learn from us! Where construction projects have been "Fasttracked" (read this "designed on the fly"), cost overruns and quality problems rival those of traditional software projects.

The implications of Architecture as an organizing principal for systems engineering work suggest we cannot afford to do our work any other way -- and stay in business.

Successful participants in the Information Systems industry will either build quality into their process, or alternatives will be found by their customers which bypass the problem.

BIBLIOGRAPHY

Alexander, Christopher,
Notes on the Synthesis of Form;
Harvard University Press, Cambridge, Mass., 1964.

Brooks Jr., Frederic P.;
The Mythical Man Month - Essays on Software Engineering;
Addison-Wesley, Reading, Mass., 1975.

DeMarco, Tom;
Controlling Software Projects - Management, Measurement and Estimation;
Yourdon Press, New York, New York, 1982.

DeMarco, Tom, and Lister, Timothy;
Peopleware;
Dorset House, NY, NY 1987.

DeMarco, Tom;
Structured Analysis and System Specification;
Yourdon Press, Englewood Cliffs, NJ, 1979.

Deming, W. Edwards;
Out of the Crisis;
M.I.T., Cambridge, Mass., 1986.

Flavin, Matt;
Fundamental Concepts of Information Modeling;
Yourdon Press, Englewood Cliffs, NJ, 1981.

Jones, Capers;
Programming Productivity - Issues for the Eighties, 2nd Ed.;
IEEE Computer Society, Piscataway, New Jersey, 1986.

Jones, Capers;
"Reusability in Programming - A Survey of the State of the Art",
IEEE Transactions on Software Engineering;
IEEE Computer Society, Piscataway, New Jersey, 1984.

Ledbetter, Lamar, and Cox, Brad;
"Software-ICs", BYTE;
McGraw-Hill, Inc., New York, New York, June 1985.

Lenzi, Marie A.
"Adopting Object Oriented Programming",
Pre-publication draft and notes;
Syrinx, Corp., NY, NY, 1989.

Orr, Kenneth;
Data Structured Systems Development Design Library;
Ken Orr & Associates, Topeka, Kansas, 1986.

Shelton, Robert E.
"Software Construction Project Management",
Pacific Northwest Software Quality Conference Proceedings;
PNSQC, Portland, Oregon, 1988;

Shlaer, Sally, and Mellor, Stephen;
"Understanding Object-Oriented Analysis",
DesignCenter Magazine, January 1989.

Walton, Mary;
The Deming Management Method;
Dodd, Mead & Company, NY, 1986.

Weinberg, Gerald M.
The Secrets of Consulting;
Dorset House Publishing, NY, NY, 1985.

Zachman, John A.;
"A Framework for Information Systems Architecture",
IBM Systems Journal, Vol. 26, No. 3, 1987.

Zultner, Richard;
"Software Quality Engineering: The Deming Way",
The American Programmer, Vol. 2, No. 6, 1989.

The Deming Way to software quality

Richard Zultner, CQE

Zultner & Company
12 Wallingford Drive
Princeton, NJ 08540
(609) 452-0216

ABSTRACT

This paper applies the *total quality management* approach of **Dr. W. Edwards Deming**, "the man who taught Japan quality," to software development.

Managers and software professionals seeking to develop higher quality software, at reduced cost, with shorter schedules, should understand the way to total quality management that Dr. Deming teaches. His **Fourteen Points** for management, his **Seven "Deadly Diseases"** for quality, and his **Obstacles** to quality can all be applied to software. These are essentially what he taught the Japanese over 30 years ago, and what he teaches managers today.

BIOGRAPHY

Richard Zultner is an international consultant, educator, writer, and speaker. He has a Master's in Management from the J.L. Kellogg Graduate School of Management at Northwestern University, and has professional certifications in quality, project management, and software engineering. He is a member of several professional societies, and as Chair of the AMERICAN SOCIETY FOR QUALITY CONTROL/SOFTWARE DIVISION'S Certification Committee, he is currently working to define a formal body of knowledge for software quality engineering.

After World War II, Japanese manufactured products had a reputation for low quality. Yet within twenty years they had reached world-class quality levels in industry after industry—with lower costs and faster production. In Japan, "Total Quality Control" had become an accepted route to competitive advantage, and the rigorous standards of the Deming Prize gave all companies a demanding benchmark for their improvement efforts. According to the Japanese, what started this transformation?

Dr. W. Edwards Deming, an American statistician, went to Japan after World War II to teach Japanese firms how to improve the quality of their export goods to pay for imported food. Japan had a devastated industrial and financial base, and no natural resources. Their reputation was for cheap and shoddy goods. Dr. Deming told Japanese managers they could produce goods and deliver services of the highest quality *faster and cheaper* by following his approach. All that was necessary was for management to learn what to do, and do it. Many did not believe him—but having lost all, had nothing to lose. So they started on the Deming Way. In 1960 Dr. Deming received the Second Order of the Sacred Treasure from the Emperor of Japan for service to the people of Japan. The rebirth of Japanese industry, and the success of their world-wide marketing efforts, were attributed to his approach.

In 1980 the NBC special *"If Japan Can...Why Can't We?"* aired, and the Deming Way was finally discovered in America. It is not just "statistical process control," or merely a "manufacturing approach," but a *total quality management approach for continuous improvement of quality.* Dr. Deming's **Fourteen Points** for management, his **Seven "Deadly Diseases"** for quality, and his **Obstacles** to quality, are essentially what he taught the Japanese over 30 years ago, and what he teaches managers today. With proper adaptation and understanding, these principles can show the way to software managers and developers working toward total quality management.

Here Dr. Deming's Way has been adapted to software development, and organized into four fundamental areas (see Figure 1):

- **people**—are the primary source of quality and productivity improvement (limited only by their knowledge)
- **statistics**—is basis for decisions based on data (and a knowledge of the nature of variation)
- **organization**—must support quality improvement and innovation
- **philosophy**—must be understood and acted upon every day

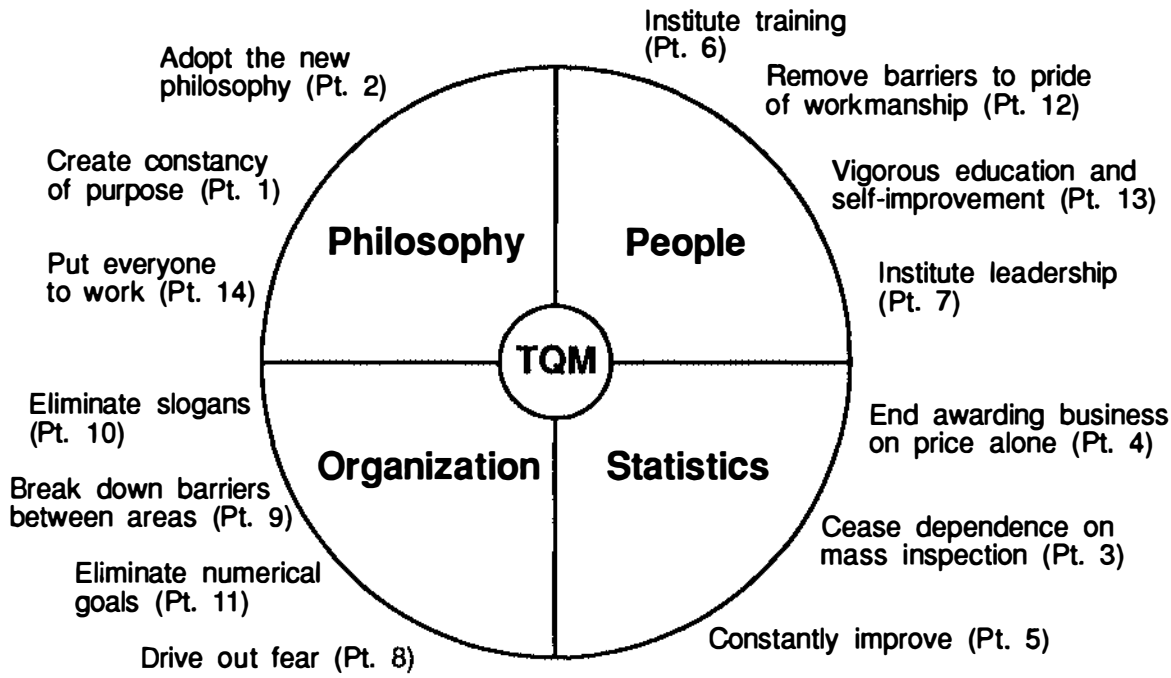


Figure 1. Dr. Demings's 14 Points for Total Quality Management

People

It is people that build software, and it is people must improve the software development process. How people are led, their skills, their knowledge, and the pride they take in their work, are critical factors in producing quality software. The initial step towards improvement begins with the proper training of new people.

Institute training (Pt. 6). Quality improvement starts with training and ends with training. An improved capability to build software better, faster, and cheaper requires improved skills of software developers and project managers. They must understand what to do in their jobs, and how to do their work. A major obstacle to quality is an elastic standard of what is acceptable or not. Too often this depends on the particular project manager and their progress against plan. **Operational Definitions** are needed for each key step in software development and for each significant deliverable component. Several good and bad examples (with explanations) must be developed and distributed to all practitioners. These then define acceptable and unacceptable in a consistent way for everyone. As people become more sophisticated, the examples and explanations must be refined. Before people are asked to "do good work," someone must define "good." Therefore, provide operational definitions of all key steps in developing and maintaining systems, and train people how to do acceptable work consistently—and then improve. There are several obstacles related to training:

> **Poor teaching of statistical methods (Ob. 5).** And software engineering. And project management. Management must insist on first-rate training for their people, and themselves. Statistics in particular is generally not taught as a set of techniques a manager can use every day. *Analytic statistics* for prediction

and planning are fundamental to the Deming Way. Traditional *enumerative statistics*, focused on the past, are not relevant—but that's what is usually taught.

Excessive maintenance costs (DD 7). Besides building higher quality systems, software groups must maintain existing systems better. The maintenance area is overdue for serious training for software engineering, project management, and quality techniques. It is *easier and faster* to improve quality in maintenance because there's a shorter learning curve—the projects are shorter and more frequent. Since many software organizations spend over half their total resources in maintenance and support, improved quality here is vital.

Once people have solid skills from good training, management must then remove the barriers to using new skills, or the training (and later education) will be wasted.

Remove barriers to pride of workmanship (Pt. 12). Software professionals are eager to do a good job; they are deeply troubled when they cannot. When software developers must temper their pride in their work with excuses ("given the time we had...with the support we got...no one really had much experience with this") quality and productivity are distant dreams indeed. Examine every management system, every policy, and every procedure to determine whether it supports or inhibits ongoing improvement. Are suggestions for improvements solicited and acted upon? This is the first challenge to management:

Excessive personnel costs (DD 6). There are too many people in software engineering doing the wrong things (often with excellent intentions) at the wrong time—over and over. Many methodologies have a significant number of steps that add only cost, delay, and hassle. How many steps actually add value for the user?

The second challenge is the annual appraisal process as it is practiced in many organizations. This may be difficult to change for a software organization (unless the rest of the organization changes too). Most organizations focus on the outputs, not the process, and pay no attention to efforts to improve the process. Since it is very difficult to measure performance, the result is an appraisal with a false degree of precision; it's unsound, and unfair. It can be very difficult to recognize an individual for teamwork, and very easy to reward someone for fighting fires (and not improving the process), even those of their own making. This is a deadly disease:

Evaluation of performance, merit rating, and annual reviews (DD 3). These are counterproductive for individuals and damage pride of workmanship according to Dr. Deming. To accurately evaluate individual performance, the influence of the boss and the work environment and first must be subtracted. Otherwise the evaluation is simply inaccurate (not to mention unfair). Inaccurate and unfair evaluations are destructive to quality and productivity. Only managers can change the process (culture and environment) which dominate any individual's performance.

Training people to develop their skills, and removing barriers to good work are first steps. It is also necessary to improve peoples' knowledge—and their ability to innovate and improve.

Institute a vigorous program of education and self-improvement for *everyone* (Pt. 13). Software managers and professionals must be educated to view quality in a new light. Organizations must realize that their people truly are their most important asset, stop treating them as commodities, and reinvest in their re-education throughout their career. Individual self-improvement should be encouraged as a way of job enrichment. Building better software faster

and cheaper requires more knowledgeable developers and managers. Such education will never end. Educate software managers and engineers to study how the software engineering process works, and how to improve it. To continuously improve a process requires profound knowledge about the nature of variation. Several obstacles work against education and self-improvement:

> **Hope for instant solutions (Ob. 1).** The desire for quick, easy answers is common in software organizations (and others!). Significant improvements in software quality or productivity are achieved only through serious study and earnest hard work. There are no jiffy solutions, and certainly none you can simply *buy*. The challenge is too substantial and complex. "There is no instant pudding," says Dr. Deming, "profound knowledge is necessary."
>
> **Obsolescence in schools (Ob. 4).** Unfortunately, many schools do not teach how *improve* the way an organization builds software—how to use quality improvement tools and techniques—or how to *prevent* problems. The emphasis must change to designing and building quality in—not sorting out the bad code by testing and reworking it. How can we delight the user?

With the base of training and education, and the removal of barriers, the capability to improve is established. But what should be improved first? Management must lead the improvement effort.

Institute leadership (Pt. 7). Leadership for quality means "quality first," and everyone must know that. Quality must be the first priority, and management must not only believe that, but *lead* their people accordingly. The quality vision of management must be shared with their people by management actions. Managers must lead—not merely manage—for quality. They must help their people to do better. The people developing and maintaining software are today already doing their best. Improving on this will take inspired leadership, not mere supervision. Software managers must study all aspects of developing, using, and maintaining software to lead their people intelligently toward improvement. For example: (1) project managers must spend time helping their people and continually proving constancy of purpose to each team member, and (2) project managers must be trained in statistical thinking so that they can find genuine causes of trouble—not chase superstitions. One of the obstacles to instituting leadership is a failure to understand the need for leadership for quality:

> **"We installed quality control" (Ob. 10).** Management may establish a quality policy, or initiate a quality program, but the task of continuous improvement in the Deming Way is never-ending. It must be a permanent long-term commitment—and demonstrated by management in actions each and every day.

Is quality the first agenda item at every management meeting? How much time is management spending on quality issues? What recent actions demonstrate management's commitment to quality? One of the clearest ways management can communicate their commitment to quality is by investing in education in *how to improve.*

The leadership, training, education, and work environment of software professionals are fundamental to the Deming Way to software quality. With these elements people have the basis for improvement. People are the key to improving any system, including the system for developing software. To actually make the improvements requires an additional element, the ability to think statistically.

Statistics

Software managers need accurate information that indicates when to take action, and when not to. They need to focus attention on those people or processes whose performance is out of (statistical) control—not those whose performance is merely low. A process in **statistical control** is stable and delivers consistent (predictable) performance, and can be improved by analyzing and then changing the process. A process that is not in control exhibits no consistent performance. The resulting variation may be so wide that no one could tell if a change improved the process or not! Therefore, first get the process in control (stable with consistent performance), then improve the capability of the process (i.e., the level of performance). In any team whose work is in (statistical) control, half will always be below average. It is those outside the control limits that management must attend to. Basic training in analytic statistics is as essential for quality, as a solid understanding of the structured tools and techniques is for software engineering. Three important areas for thinking statistically are testing, purchasing, and improving software development.

End the practice of awarding business on price tag alone (Pt. 4). The goal is to minimize the total lifetime costs—not the purchase price. Find the best quality vendor and work with them through multi-year contracts to lower the long-term costs. A single-source relationship for a particular product or service (with a continuously improving vendor) will result in consistency, uniformity, and reliability—all essential to quality. The price of a product or service has no meaning without an associated measure of quality. Without such measures, business sinks to the lowest bidder, with low quality and high total cost the inevitable result. A common purchasing-related obstacle to quality is:

> **The belief that new hardware or packages will transform software development (Ob 2).** Buying automated tools or using "state-of-the-art" packages can't possibly yield the benefits hoped for if they aren't used wisely. First understand, stabilize, and improve your software engineering process—then buy the latest tools. Only then will you be able to use them to full advantage.

It is also important to understand that if the package isn't user-modifiable, then it can become a barrier to further improvement. If an area of your organization is automated, and can't easily be changed, how will you improve it? Continuous improvement of all key business and software engineering processes is fundamental to the Deming Way.

Cease dependence on mass inspection—*especially testing* (Pt. 3). The traditional thrust of software quality has been to use brute-force testing. Yet testing neither improves nor guarantees quality. Testing simply (and imperfectly) sorts code into two piles: "OK so far," and "rework." Rework adds delay and cost, but no value. Many project plans (and even methodologies) call for a massive effort to (try to) find and fix the large number of defects expected—even taking up an entire phase of the project to do so. What does this say to the developers about how many errors are expected? Is their job to build correct software—or meet the schedule? This attitude is one obstacle to quality:

> **"That's good enough—we don't have time to do better" (Ob. 6).** This an open admission that it's okay to make (an acceptable number of) defects; they'll be caught in testing. However, it is faster and cheaper to concentrate effort upstream and detect (and correct) defects early—or even better, *prevent* the defects from occurring.

Although testing doesn't add value, it is necessary to test, and to do so cost-effectively. One difficulty with testing is that it is often not well done. Too much of the wrong kind of testing is often done at the wrong time. And if testing is done without the expected results of the tests being made, it is unlikely that any subtle errors could be found. Testing can be much more effective, even with a reduced emphasis, with a carefully planned and rigorously executed program of structured testing (requiring in turn rigorous functional and technical specifications). One specific testing obstacle concerns prototypes:

> **Inadequate testing of prototypes (Ob. 14).** Proper testing and experimentation requires an understanding of analytic statistics (as opposed to traditional enumerative statistics) and independent inspection techniques. More attention needs to be given to finding out what features are expected (and thus not mentioned), and which are unimagined but desired (and thus not mentioned), and delightful when delivered.

Substantially more effort must be spent up front during analysis and design to catch defects soon after they're made. Inspections—or rigorous (structured) walkthroughs—should to be much more frequent during analysis and design. These inspections should be **independent inspections**—not "inspection by collusion," as Dr. Deming calls it. Each person selected to participate in an inspection must independently prepare a list of errors in the work product. Selection of participants from among those qualified should be completely random to avoid bias. The appraisal therefore, actually occurs during the preparation for the inspection. Operational definitions make this possible. The inspection meeting then serves as a forum for comparing the error lists. This permits a determination of whether the inspectors are in agreement on identification of errors, and therefore whether the appraisal process itself is in (statistical) control and improving.

Dr. Deming teaches that the goal to work toward is to get key development processes under statistical control so they produce good results the first time. Then testing may be reduced—but is still needed for the purpose of gathering quality metrics. Another area where statistical thinking is needed is purchasing.

Constantly and forever improve the software development process (Pt. 5). Managers must insist on ongoing improvement by everyone—including themselves. This means the software engineering process must be defined and then improved. The system development methodology, standards, and procedures should be revised on an ongoing basis. Whatever a software organization knows about building software should be reflected in their formal approach (their methodology) and improved at least each year—according to a plan for improvement. Are you any better at building systems this year than last year? Than five years ago? Are the same mistakes being repeated? What will change such that next year will be better? What is the plan to direct this improvement? Are you getting better at improving, and planning improvement? One obstacle to getting assistance in improving is:

> **"Anyone that comes to help us must understand all about our systems" (Ob. 15).** A knowledgeable, objective, third party can make substantial contributions. All world-class athletes have experienced coaches to objectively advise them and help them improve. The coach has to be an expert coach, not a great athlete.

Improvement requires increased knowledge—there is no substitute. The difference between world-class systems developers and others isn't the tools and equipment—it's their knowledge. Their *process* of systems development is superior. This knowledge is reflected in their greater effort up-front, and a deemphasis on testing. Teamwork among all parties begins in analysis and design and continues through use and maintenance. Software groups must organize for quality to get higher quality software.

Organization

The organizational environment for software development is an critical factor in producing better software, faster and cheaper. To improve, software organizations must enhance their development environment to promote quality work. Only management can abandon slogans, break down departmental barriers, and eliminate plan-less numerical goals. These are major points for management to work on.

Eliminate slogans, exhortations, and targets (Pt. 10). Software managers must not merely encourage *others* to improve. Everyone is already doing their best. Management must lead the way. Most problems arise from the process, not the people. And only management can change the process. Slogans and targets for improved quality or productivity without a plan to meet them simply advertise management's ignorance of the real barriers to quality. Let the developers put up their own signs and slogans when they see their managers leading them to improved quality. The goal is continuous improvement, and the attention of management should be on ongoing efforts to improve the software engineering process—not on hitting some arbitrary target on some magical date. One example of an obstacle to this is:

> **The unmanned computer (Ob. 11).** Computer Aided Software Engineering (CASE) packages and sophisticated software environments won't significantly improve productivity without a solid understanding of the underlying software engineering techniques. Automating what is unclear and misunderstood simply locks ignorance into the organization. First gain the necessary knowledge to steadily and consistently improve at developing systems. Learn, simplify, then automate. What software development problems will the CASE package solve, and to what extent? And how will it solve those problems?

Quality improvement requires more than pushing money or words at the problem. Instead, management must take the lead in: (1) *planning* for quality improvements, (2) *doing* what the plans call for, (3) *checking* to see what results the changed process delivered, and (4) *analyzing* what occurred, and *acting* on that analysis. Software professionals, the real experts on the software engineering process, should be heavily involved in studying their process, recommending improvements, and carrying out the approved changes. This will require cooperation across departmental boundaries.

Break down barriers between areas (Pt. 9). Software managers and project teams must learn about the problems upstream and downstream from their activity. Everyone has a *customer* and a *supplier*. Spend time working with "suppliers" to improve inputs, and asking "customers" how outputs should be improved. People in maintenance groups can be a rich source of information for improvement. A particular problem for software managers is making sure that one task (such as the analysis of a subsystem or design unit) is truly finished before starting the next task (such as design). One obstacle to applying quality techniques developed outside of software is:

> **"Our problems are different" (Ob. 3).** Most problems are caused by the *process* of developing systems, and that process has much in common with service processes and modern manufacturing processes. With appropriate adjustments and adaptations, almost all quality techniques can be applied to software. For example statistical process control (SPC) and quality function deployment (QFD) can be applied to any process—including software development.

For software, there is often a bilateral relationship between customers and suppliers. For example, a Data Administration group may supply data dictionary support to the project team (needed to capture detailed data definitions), and in return are supplied completed data definitions (needed to manage information as a corporate resource). Every *stakeholder* involved in a project must understand the role they play, and others play. All should work toward customer satisfaction and the good of the organization as a whole. This means they must not be afraid to voice their concerns.

Drive out fear (Pt. 8). People must feel secure, or they will not ask questions, or request help—even when they do not understand their tasks. Don't blame the professional staff for problems that only software managers have the power to remedy. People must feel comfortable to master the work they are to do, and make suggestions for ongoing improvement. Managers must act as a coach—not a judge (and not punish for poor quality). One obstacle is the belief that *others* are always at fault:

> **"Our troubles lie entirely with the programmers" (Ob. 8).** Management hires them (carefully or carelessly), trains them (or not), and tells them "meet the schedule (no matter what)" and then complains about user complaints. How do managers know the programmers are to blame? Dr. Deming holds that about 90% of the quality problems are inherent in the development process—and only management can change that.

This is a most difficult point. Managers often cannot tell the degree to which fear rules their area. If people are afraid to admit a mistake, or worried they may not have an answer when asked, then fear is present. People are often afraid of knowledge; it may reveal shortcomings. Yet knowledge is a key for improvement. Fear may also drive people to do things they know to be silly or wrong, resulting in dissatisfied users and a low respect for management. A lack of understanding of variation can lead managers to tamper with a stable process, sowing misunderstanding, mistrust, and fear—destroying quality. An important step in reducing fear is to avoid the misuse of numerical goals.

Eliminate numerical quotas and goals (Pt. 11). Software professionals, to keep their jobs, may foolishly meet their current schedule at any cost—without regard to the ensuing damage to users from poor quality. This assures inefficiency and high total cost. A schedule that causes haste and defects, instead of quality, accomplishes nothing and serves no one. Concentrate on helping people do a better job by reducing rework, errors, and waste. Work toward ongoing improvement, not achieving some arbitrary, short-term goal—that's an obstacle:

> **Emphasis on short-term schedules (DD 2).** If quality isn't more important than the deadline for most projects, then the organization simply isn't serious about quality. Better late than wrong. Management actions must communicate this every day. First invest the time to learn how to do it right, then work to improve the process (and get faster and cheaper). Speed follows inevitably as the work is understood and streamlined—and the amount of rework declines significantly. The result is a sustainable improvement.

Goals dictated to others, without the means to achieve them, are destructive. A goal without a realistic plan to achieve it is dangerous. If a project manager and team do not understand how to meet a deadline (which the project plan should clearly show) they will concentrate on just meeting the date, not on satisfying the user. Everyone must understand that software groups should build and maintain systems not to hit dates, but to satisfy users and support the organization. Junk is worthless—whether on-time or not. This point is often misunderstood. It is the plan, not the goal that is important. The plan is the path to improvement. A plan-less goal usually reflects management's lack of knowledge—or "management by fear." If the system development process is stable (in statistical control) it will deliver what it is capable of—regardless of the goal. If it's not stable, then no one has any sound basis for setting a goal. A goal beyond the ability of the process won't be achieved. And not all goals lend themselves to numbers:

> **Managing by "visible figures" alone (DD 5).** Everyone wants increased productivity. Increasing software productivity requires building smarter, and that depends on developing a deeper understanding of systems development. This is better accomplished by trying to maximize quality (i.e. do a better job), rather than by trying to minimize schedule, budget, or resources. Most software managers' reports don't measure the really important things, like *user satisfaction*. Don't allow mere numbers to dominate business decisions (especially the wrong ones).

It is worthwhile to try to develop the best quantitative measures you can, for all important software development steps. This is fundamental to applying statistical process control (SPC) and quality function deployment (QFD) to software. And such measures must be refined and improved over time. But only after specific tasks are well understood can they be accurately measured and improved to get greater speed or reduced effort. High quality software takes less time and requires fewer resources than poor quality software. Concentrating first on increased speed or reduced cost leads to cutting corners and only short-term productivity gains. Any corners cut are paid for dearly later. Improve quality, and productivity always improves, and stays improved. This is a fundamental point of management philosophy in the Deming Way.

Philosophy

To achieve the results of building better systems faster and cheaper, management must understand what to do, and do it. Their philosophy must be sophisticated, and put quality first.

A change is needed in management thinking. The old philosophy delivered the old results. New, better results are required. That calls for a new philosophy.

Adopt the new philosophy (Pt. 2). Traditionally accepted levels of defects in requirements, design, code, and maintenance can no longer be tolerated. Errors and mistakes require rework, which causes delay and increases cost. The users and customers ultimately pay for this—and they can't afford it. A common obstacle is a misunderstanding about specifications:

> **The belief it is only necessary to meet specifications (Ob. 12).** Just meeting the current specifications isn't enough. Management must work to make the system development process even more capable—able to deliver superior performance next time. Future specifications will be even more demanding. Software organizations must improve at reflecting true user requirements and expectations in software specifications.

> The users' expectations are always increasing and expanding with every successful system they see. They become ever more sophisticated users of software.

The goal isn't to simply meet specifications, it is to deliver software so superb, that users' expectations aren't met—but exceeded. Users should not just be satisfied—but delighted. The software organization must become an organization that strives to succeed at this never-ending task. An additional obstacle is:

> **The fallacy of zero defects (Ob. 13).** It is simply not enough to build software with nothing wrong. The software must also do everything right. Software developers must make sure they include the expected features—the absence of which would dissatisfy users. They must include the normal features—which if poorly implemented would disappoint, and if well implemented would satisfy users. And they must include exciting, unexpected features—based on a deep understanding of the purpose of the system, which excite and delight users. Delivering the wrong system which works perfectly is not going to satisfy users.

A transformation of software management is required—with inspired leadership for quality. Most of the quality problems in systems development can only be solved by managers. By words and deeds, make it clear that quality comes first, that everyone must improve.

Create constancy of purpose for the *improvement* of software and service (Pt. 1). Software managers must address the problems of today and tomorrow. The problems of today are problems in the short run, meeting the next milestone, fixing the latest bug, and so on. The problems of tomorrow require constant focus on improvement—to serve users better tomorrow than today. Software managers must dedicate themselves to: (1) *constantly improve* in the analysis and design of systems. Understanding *why* the users have their problems and opportunities, *what* the users and want and expect, and then determining *how* to do that, is the key to satisfying users. Becoming more efficient at coding and testing will not lead to satisfied users. (2) invest in *education and research.* Software is built by people. To improve, software managers and engineers must have a deeper knowledge of their field—and quality—through formal training. They must learn of new developments and try them on their projects. Software groups must develop organized ways to search out improvements and breakthroughs in systems development, use, and maintenance; and to disseminate these new approaches in their organization. (3) *innovate* for better systems and services in the future. Users want better systems, built in less time, with fewer resources. This requires changing the systems development process by constantly experimenting with new techniques and mastering them. There are several obstacles to this:

> **Lack of constancy of purpose (DD 1).** Software quality won't improve if it's seen as a fad. The commitment must be permanent and unwavering. Quality must be priority one—for years.

> **False starts (Ob. 9).** Unfortunately, it is quite possible for managers to misunderstand the underlying statistical thinking behind the Deming Way, and drive their software organization to ruin trying to follow their interpretation. Management must study and understand statistical thinking. It is also not possible to pick and choose which Points to follow. All follow from statistical theory.

A particular problem is improving the software organization in the face of high turnover. This is a serious problem for constancy of purpose, as the "organizational wisdom" is constantly being diluted and dissipated:

> **Mobility of systems professionals and managers (DD 4).** Mobility saps the perseverance of the organization, and encourages a short range focus. High turnover is not inevitable in a software group. Create an a superb environment for systems development, and people will create superb systems—and stay.

Put everyone to work to accomplish the transformation (Pt. 12). No one is exempt from the task of improvement—especially management. Top software managers must formulate a quality policy, plan a quality program, and carry out the quality mission *personally*. A critical mass of people in the organization must deeply understand the Fourteen Points, the Seven Deadly Diseases, and the Obstacles. An obstacle to this point is the mistaken notion that only some people must work on improvement:

> **"Our quality control people take care of all our quality problems" (Ob. 7).** Quality is not the job of the quality assurance group, or the testing group, it is the job of the development group. Or "perhaps the job of the developers is to produce defects so the testers have a job to do?" asks Dr. Deming. Appraisal (inspections, reviews, testing, etc.) deals with errors after they have occurred. The emphasis must shift to the developers learning to prevent errors. Modern software quality techniques, such as Quality Function Deployment, are aimed at this goal.

The transformation is not quick. It is not easy. Usually it is the hardest thing ever attempted by the organization, and they will work harder than they have ever worked on anything, for longer than they have ever worked on anything. But that is what's required to become a world-class software organization. And there is no finer place for a software engineer or manager, than working with the best people in the world.

Conclusion

The Deming Way provides a guide to total quality management for software managers and professionals. Adapted to software development and organized into four fundamental areas: people, statistics, organization, and philosophy; Dr. Deming's Fourteen Points, Seven Deadly Diseases, and Obstacles show us what to do (and what not to do) to improve continuously. Anyone seeking to produce higher quality software at reduced cost and shorter schedules should seriously study the Deming Way to software quality.

ACKNOWLEDGEMENTS

I am indebted to many people for the development of the ideas expressed in this paper. Chief among them are W. Edwards Deming, Brian Joiner, and Heero Hacquebord, who encouraged me to apply their approach to software. I would also like to thank Bob King, Bill Scherkenbach, and Peter Scholtes for their valuable insights.

REFERENCES

Especially recommended are Dr. Deming's *Out of the Crisis*, for its complete coverage of the Deming Way, and Scherkenbach's *The Deming Route to Quality and Productivity* to further explain Deming's insights.

Deming, W. Edwards. 1986. *Out of the Crisis*. Cambridge: MIT Center for Advanced Engineering Study. ISBN 0-911379-01-0

Gitlow, Howard S., and Shelly J. Gitlow. 1987. *The Deming Guide to Quality and Competitive Position*. Englewood Cliffs: Prentice-Hall. ISBN 0-13-198441-1

Imai, Masaaki. 1986. *Kaizen: The Key to Japan's Competitive Success*. New York: Random House. ISBN 0-394-55186-9

Ishikawa, Kaoru, and David J. Lu, trans. 1985. *What is Total Quality Control? The Japanese Way*. 2nd rev. ed. Tokyo: Asian Productivity Organization. ISBN 0-13-952433-9

Mann, Nancy R. 1985. *The Keys to Excellence: The Story of the Deming Philosophy*. Los Angeles: Prestwick Books. ISBN 0-9614986-0-9

Scherkenbach, William W. 1986. *The Deming Route to Quality and Productivity: Road Maps and Roadblocks*. Milwaukee: ASQC Quality Press. ISBN 0-941893-00-6

Walton, Mary. 1986. *The Deming Management Method*. New York: Dodd, Mead & Company. ISBN 0-396-08683-7

❄

APPENDIX

The 14 Points for software managers

1. **Create constancy of purpose for the *improvement* of software and service,** with the aim to become excellent, satisfy users, and provide jobs.
2. **Adopt the new philosophy.** We are in a new age of software engineering and project management. Software managers must awaken to the challenge, learn their responsibilities, and take on leadership for change.
3. **Cease dependence on mass inspection *(especially testing)* to achieve quality.** Reduce the need for inspection on a mass basis by building quality into the software in the first place. Inspection is not the answer. It is too late and unreliable—it does not produce quality.
4. **End the practice of awarding business on price alone. *Minimize total cost.*** Move toward a single supplier for any one item or service, making them a partner in a long-term relationship of loyalty and trust.
5. **Constantly and forever improve the software development process,** to improve quality and productivity, and thus constantly decrease the time and cost of software. Improving quality is not a one time effort.
6. **Institute training on the job.** Everyone must be well trained, as knowledge is essential for improvement.
7. **Institute leadership.** It is a manager's job to help their people and their systems do a better job. Supervision of software managers is in need of an overhaul, as is supervision of professional staff.
8. **Drive out fear, so that everyone may work effectively.** Management should be held responsible for faults of the organization and environment.
9. **Break down barriers between areas. *People must work as a team.*** They must foresee and prevent problems during software development and use.
10. **Eliminate slogans, exhortations, and targets that ask for zero defects, and new levels of productivity.** Slogans do not build quality software.
11. **Eliminate numerical quotas and goals. *Substitute leadership.*** Quotas and goals (such as schedules) address numbers—not quality and methods.
12. **Remove barriers to pride of workmanship.** The responsibility of project managers must be changed from schedules to quality.
13. **Institute a vigorous program of education and self-improvement *for everyone.*** There must be a continuing training and education commitment by software managers and professional staff.
14. **Put *everyone* to work to accomplish the transformation.** The transformation is everyone's job. Every activity, job, and task is part of a process. Everyone has a part to play in improvement.

The 7 Deadly Diseases for software quality

1. **Lack of constancy of purpose** to develop software that will satisfy users, keep software developers in demand, and provide jobs.

2. **Emphasis on short-term schedules**—short term thinking (just the opposite of constancy of purpose toward improvement), fed by fear of cancellations and layoffs, kills quality.
3. **Evaluation of performance, merit rating, and annual reviews**—the effects of which are devastating on individuals, and therefore, quality.
4. **Mobility of software professionals and managers**. Job hopping makes constancy of purpose, and building organizational knowledge, very difficult.
5. **Managing by "visible figures" alone**—with little consideration of the figures that are unknown and unknowable.
6. **Excessive personnel costs.** Due to inefficient development procedures, stressful environment, and high turnover, software development person-hours are too high.
7. **Excessive maintenance costs.** Due to bad design, error ridden development, and poor maintenance practices, the total lifetime cost of software is enormous.

The Obstacles to software quality

1. **Hope for instant solutions.** The only solution that works is knowledge—solidly applied, with determination and hard work.
2. **The belief that new hardware or packages will transform software development.** Quality (and productivity) comes from people, *not* fancy equipment and programs.
3. **"Our problems are different."** Software *quality* problems simply aren't unique—or uncommon.
4. **Obsolescence in schools.** Most universities don't teach software quality—just appraisal techniques.
5. **Poor teaching of statistical methods.** Many software groups don't have good statistically-oriented training in quality or project management.
6. **"That's good enough—we don't have time to do better"**—but time *will* be spent later to fix the errors. Doing the right things right the first time (and every time) is fastest.
7. **"Our quality control people take care of all our quality problems."** Quality is management's responsibility, and cannot be delegated. Either management does it, or it doesn't happen.
8. **"Our troubles lie entirely with the programmers."** Who hired the programmers? Trained them (or not)? Manages them? Only management can do what must be done to improve.
9. **False starts** with quality (or productivity). Impatient managers who don't understand that quality is a long term proposition quickly lose interest.
10. **"We installed quality control."** Quality is a never-ending *daily* task of management. Achieve consistency (statistical control)—then continuously improve.
11. **The unmanned computer**—such as a CASE package used without a solid knowledge of software engineering.
12. **The belief it is only necessary to meet specifications.** Just meeting specifications is not sufficient. Continue to improve consistency and reduce development time.
13. **The fallacy of zero defects.** Constant improvement doesn't end with zero defects (all specs met). The mere absence of defects is no guarantee of user satisfaction.
14. **Inadequate testing of prototypes.** The primary purpose of testing prototypes is to *learn*—and then apply that knowledge to a robust production system.
15. **"Anyone that comes to help us must understand all about our systems."** Software managers may know all there is to know about their systems and software engineering—except how to improve.

Software Density Metrics: Assumptions, Applications, and Limitations

Tze-Jie Yu

AT&T, Bell Laboratories
Naperville, IL 60566

ABSTRACT

This paper discusses the applications and limitations of several software size-normalized metrics, including software fault density, complexity density, and inspection rate. The focus of this study is on the assumptions behind these size-normalized metrics, and whether or not these assumptions are met by empirical data. The conclusion is that most empirical data does not support the assumptions, and the use of density metrics may lead to improper modeling and analysis. General approaches to the metric data analysis are suggested as an alternative to the use of size-normalized metrics.

BIOGRAPHY

Tze-Jie Yu received the B.S. degree in Electrical Engineering from National Taiwan University, Taipei, Taiwan, Republic of China, in 1979, and the M.S. and Ph.D. degrees in Computer Science from Purdue University, West Lafayette, Indiana, in 1983 and 1985 respectively. His thesis work was on the software fault and reliability models based on the data collected from several industrial companies.

Dr. Yu joined AT&T Bell Laboratories, Naperville, Illinois as a Member of Technical Staff in 1985. He is currently developing software tools, metrics, and models to control, predict, and improve software quality and productivity of the 5ESS® Switch for international applications. He was involved in system testing and performance evaluation of the 5ESS Switch.

Dr. Yu has published several papers and many technical reports in the areas of software metrics, software fault and reliability models, programming techniques, software testing, software tools, code inspection, and switching performance measurements. He is a member of the IEEE Computer Society.

1. INTRODUCTION

The purpose of this paper is to discuss the applications and limitations of a widely used metric, software fault density metric, which is software faults divided by program size. The discussion can be applied to several other size-normalized metrics, such as complexity density (a complexity metric divided by program size), and inspection rate (program size divided by inspection time).

The approach of this study is first to identify the intended goals (or applications) of a density metric. Based on the goals, we identify the assumptions of the metric and collect data to check whether or not the assumptions are met. We also study the problems of applying a density metric when its assumptions are not supported by empirical data.

2. THEORETICAL BASIS

Given a set of programs or a set of modules from a large software system,

$$P = (p_1, p_2, \cdots p_n)$$

we can compute software metrics, such as program size and software faults, from these programs. Given two metrics X and Y, we can collect two data sets for the metrics:

$$(x_1, x_2, \cdots x_n)$$

and

$$(y_1, y_2, \cdots y_n)$$

The first step of data analysis is to determine their measurement scales,[1] where measurement scales are

nominal — The data from this scale is used for identification and classification. The operation that is meaningful with this scale is *equality* ($x_i = x_j$ or $x_i \neq x_j$). Examples are social security number, software fault type, and software product type.

ordinal — If an ordering is possible, the data is from the ordinal scale. The operation with this scale is *ranking* ($x_i > x_j$). Examples are severity of software faults and programmer experience.

interval — When intervals, or differences, are meaningful, the data is from the interval scale. The operation with this scale is *difference* or *subtraction* ($x_i - x_j$). Examples are temperatures in Fahrenheit and calendar date.

ratio — The ratio scale is the most desirable scale for data analysis because it has the properties of other measurement scales. The operation with this scale is *ratio* or *division* ($\frac{x_i}{x_j}$). An important characteristic of data from the ratio scale is the existence of the absolute zero. Examples are temperatures in Kelvin and product sale in dollars.

The data from the ratio scale is also from the interval scale which is in the ordinal scale which is in the nominal scale.

The use of density metrics (metrics *divided* by program size) requires the data to be from the ratio scale. If we collect data from many small programs where most of the programs have no or only one fault, we would not be able to *rank* the programs based on software faults because of too many ties. In such a case, we cannot justify the data to be from the ordinal scale, let alone from the interval or ratio scale. As a result, the use of fault density would not be appropriate, regardless of the applications. In the following discussion, we *assume* all the metric data are from the interval scale. This assumption is appropriate in the context of the discussion of density metrics, while it is not appropriate for general software metric data analysis.

If we can justify that the data of X and Y are from the interval scale, we can compute the following statistics:

$$\overline{x} = \sum \frac{x_i}{n}$$

$$s_X = \left(\frac{\sum (x_i - \overline{x})^2}{(n-1)} \right)^{0.5}$$

In order to study the relationship between X and Y, we usually use the Pearson correlation coefficient

$$\text{Pearson Correlation Coefficient } (r_p) = \frac{1}{n-1} \sum \left(\frac{x_i - \overline{x}}{s_X} \times \frac{y_i - \overline{y}}{s_Y} \right)$$

The value of r_p is between 1 and -1. If $r_p > 0$, X is positively correlated with Y; i.e., Y increases as X increases. If $r_p < 0$, X is negatively correlated with Y; i.e., Y increases as X decreases. If $r_p = 1$, X and Y have a perfect positive linear relationship. If $r_p = -1$, X and Y have a perfect negative linear relationship. If $r_p = 0$, X and Y are independent of each other.

Some basic statistical properties of r_p are given as follows:

$$r_p(X, Y) = r_p(Y, X) \qquad \text{Rule-1}$$

$$r_p(a + X, Y) = r_p(X, Y) \qquad \text{Rule-2}$$

where a is a constant.

$$r_p(b \times X, Y) = r_p(X, Y) \qquad \text{Rule-3}$$

where b is a constant and greater than zero.

Combining Rules 1, 2 and 3, we can get

$$r_p(a + b \times X, c + d \times Y) = r_p(X, Y) \qquad \text{Rule-4}$$

where a, b, c, and d are constants, and b, d are greater than zero.

3. SOFTWARE FAULT DENSITY

Software fault density is defined as

$$\text{fault density}(D) = \frac{\text{faults}}{S}$$

where S is program size measured in non-commentary source lines (NCSL).

The traditional applications of the software fault density metric are in the following two areas:

- comparison and evaluation of software quality, and
- prediction of software faults.

The intended goal of applying the fault density metric is to eliminate the variation in software faults associated with program size so that evaluation and prediction of software quality are possible among programs with different size. A simple approach is to assume that software faults are *uniformly* distributed in each program, and this is the basis of fault density. The assumption of uniformity implies the linear relationship between software faults and program size. If the linear relationship does not hold, we should not use fault density to evaluate and predict software quality. Fortunately, most empirical data does support this assumption. The Pearson correlation coefficient between software faults and program size can be found in the following studies:

- A study at the subsystem level of four Bell Labs' software products shows that the Pearson correlation coefficient is between 0.8 and 0.9, where the size of each subsystem is between 0.5 KNCSL and 80 KNCSL and each product has 30 – 40 subsystems.
- According to Lipow's study of 25 modules of a software product,[2] the Pearson correlation coefficient between software faults and program size is 0.96.
- A study of 31 Algol W procedures by Schneidewind and Hoffmann shows the Pearson correlation coefficient between software faults and program size is 0.60.[3]

In addition to linearity, we often ignore another assumption behind fault density, which is the proportionality between faults and program size, i.e.,

$$\text{faults} \approx b \times S \tag{1}$$

And fault density becomes

$$\text{fault density} = \frac{faults}{S} \approx b \tag{2}$$

If Eq. (2) holds, we can evaluate program quality based on the value of proportionality constant b; a higher value of b implies poorer quality. However, most empirical data of software faults exhibits a different linear relationship:

$$\text{faults} \approx b_0 + b_1 \times S \tag{3}$$

where b_0 is significantly different from (usually greater than) zero. As a result, fault density becomes

$$\text{fault density} \approx \frac{b_0}{S} + b_1 \tag{4}$$

Based on regression analysis, Eq. (1) assumes that the regression line goes through the origin, which is a special case of Eq. (3).

From Eq. (4) the total variation in fault density is primarily dependent on the reciprocal of program size, which implies smaller programs will have higher fault density than larger programs. This counter-intuitive result is supported by fault data from a large software product shown in Table 1. Note that the size of modified files is measured by NCSL of modified and inserted code, excluding the base code.

Table 1
Software Fault Density and Program Size

Size in NCSL	New File		Modified File	
	sample	fault density	sample	fault density
0 ~ 50	1541	3.97	2110	12.7
50 ~ 100	415	2.21	149	5.65
100 ~ 200	270	1.33	78	3.36
200 ~ 400	117	1.99	31	1.82
> 400	51	1.25	29	1.79

The finding of larger programs with lower fault density has also been reported in several other papers.[4] [5] [6] [7]

To further investigate this phenomenon, we compute the Pearson correlation coefficient between fault density (D) and program size (S) as follows:

$$\begin{aligned} r_p(D, S) &\approx r_p(\frac{a + b \times S}{S}, S) \\ &= r_p(\frac{a}{S} + b, S) \\ &= r_p(\frac{1}{S}, S) < 0 \end{aligned}$$

This negative value of r_p indicates that fault density decreases as program size increases, which is consistent with the finding from empirical data.

In addition to the problem of evaluating program quality, fault density may have problems in software fault modeling. A general software fault model can be represented as follows:

$$\text{faults} = \text{f}(\text{ size, other attributes })$$

As discussed before, size is usually a dominant factor in determining the variation in faults, but size alone can explain only 40% to 80% of the total variation in faults. In order to eliminate the variation associated with program size, we resort to fault density and hope the model becomes

$$\text{fault density} = \text{f1}(\text{ other attributes })$$

However, the variation in fault density is usually dominated by program size, and the model really is

$$\text{fault density} = \text{f2}(\ 1/S, \text{other attributes})$$

The new model f2 is more complex than the original model f, and cannot provide better applications for fault prediction.

However, the fault density metric is not completely useless. If program size is large enough, the term b_0/S in Eq. (4) will be small enough to be negligible. In this case, fault density may offer a high level view of program quality and its simplicity makes it easy to apply and understand.

In summary, this study recommends starting data analysis with a general approach, such as regression analysis of size and software faults. If the Pearson correlation coefficient is significantly greater than zero and the regression line is close to the origin, fault density may be used as a quality metric or a fault predictor.

4. SOFTWARE COMPLEXITY DENSITY

In the context of this paper, a complexity metric is defined as a metric that can be computed from software programs directly. Although this definition includes the size metric as a complexity metric, the focus is on the search of *size-independent* metrics. Three complexity metrics are discussed in this section:

$v(G)$ McCabe cyclomatic complexity metrics[8]. McCabe applied graph theory to the program control flow diagram and defined the cyclomatic complexity metric, which is equivalent to the count of number of conditional statements (such as **for**, **while** and **if**), number of boolean conditions (such as **and** and **or**), and number of procedures in a program.

η_2 Unique operands of Halstead's Software Science metrics,[9] where an operand is a program variable, a constant, or a literal string.

N_{ss} Program token counts of Halstead's Software Science metrics, where a token is an operand or an operator. Examples of program operators are program delimiters, parenthesis, mathematical operations, and key words.

The discussion can be generalized to many other complexity metrics. A previous study shows that most software complexity metrics are strongly correlated with program size.[10] Metric data from a Bell Lab product with several thousand files also shows that the Pearson correlation coefficients between program size (S) and any of these three complexity metrics are greater than 0.90. As a result, these *size-dependent* metrics cannot help in increasing the predictive capability of software fault models. In search of size-independent metrics, software engineers sometimes resort to complexity density metrics, such as token density (N_{ss}/S), operand density (η_2/S), and $v(G)/S$. The intended application of complexity density metrics is to increase the predictability of a software fault model, such as

$$\text{faults} = \text{g (size, complexity density, other attributes)}$$

Before applying a complexity density metric in a software fault model, software engineers should first study the relationship between complexity densities and program size as shown in Table 2.

Table 2
Complexity Density and Program Size

Size in NCSL	$v(G)/S$	η_2/S	N_{ss}/S
0 ~ 50	0.13	0.84	4.69
50 ~ 100	0.17	0.74	4.32
100 ~ 200	0.18	0.71	4.56
200 ~ 400	0.19	0.62	4.67
400 ~ 800	0.20	0.47	5.03
> 800	0.22	0.41	4.90

Because these complexity metrics (not their density metrics) are highly correlated with program size (S), we can apply linear regression as follows:

Case-I

For the metric $v(G)$, the regression model is

$$v(G) \approx a_1 + b_1 \times S$$

where $a_1 < 0$ based on Table 2. As a result, the following model

$$\text{faults} = \text{g1 (} S, \frac{v(G)}{S}, \text{ other attributes)}$$

is similar to

$$\text{faults} = g1^* \left(S, \frac{-1}{S}, \text{other attributes} \right)$$

Case-II

For the metric η_2, the regression model is

$$\eta_2 \approx a_2 + b_2 \times S$$

where $a_2 > 0$ based on Table 2. As a result, the following model

$$\text{faults} = g2 \left(S, \frac{\eta_2}{S}, \text{other attributes} \right)$$

is similar to

$$\text{faults} = g2^* \left(S, \frac{1}{S}, \text{other attributes} \right)$$

Case-III

For the metric N_{ss}, the regression model is

$$N_{ss} \approx a_3 + b_3 \times S$$

where $a_3 \approx 0$ based on Table 2. As a result, we do have a size-independent metric N_{ss}/S. However, the metric of token counts (N_{ss}) is actually another *size* metric, and its variability is primarily associated with programming styles for a given programming language. If a software development environment has a coding standard, this metric is expected to have very small variability as shown in Table 2. Therefore, N_{ss}/S is not useful in the application of software fault modeling.

In summary, the use of software complexity metrics does not seem to be a good approach to software fault modeling. Previous studies of complexity density metrics in software fault modeling did not yield useful results.[11] [12]

Another popular study of complexity density is to set a threshold value, such as

$$\frac{v(G)}{S} < X \qquad \text{where X is the threshold value}$$

If a program is above this limit, it is considered more complex and tends to be more error-prone. Although this approach sometimes works, a more general technique is to use simple linear regression with 95% confidence interval to identify outliers. The use of density metrics is a special case of linear regression, which assumes the regression line goes through the origin.

From an empirical prospective, we recommend the following requirements for new software complexity metrics:[1]

---

1. Essential properties from a theoretical prospective for new complexity metrics can be found in [13].

- A complexity metric should be correlated with a *dependent* variable, such as programming effort, software faults, effort of reading and understanding programs, effort of fixing a fault, rating of maintainability, or other quantifiable characteristics of programs.
- The metric should not be strongly correlated with size.
  If a complexity metric is strongly correlated with size, the metric will not help in modeling the dependent variable after considering the size metric. However, if the metric data can be collected earlier than the size data, this metric will be useful in predicting the size.[14] That is, the availability is also an important issue in determining the applications of a complexity metric.

5. FAULT DENSITY AND INSPECTION RATE

Another improper use of density metrics is in modeling the code inspection process, where a general inspection model can be represented as

$$\text{code inspection faults} = \text{h (size, inspection time)}$$

The focus of the analysis is usually on the relationship between the cost (inspection time) and the benefit (identified faults). Because code inspection faults and inspection time are both linearly correlated with program size, we are interested in eliminating their variations associated with program size. An alternative model may be stated as follows:

$$\text{inspection fault density} = \text{h1 (inspection rate)}$$

where inspection rate is program size divided by inspection time. However, the variation in inspection fault density is often dominated by the reciprocal of program size, and the variation in inspection rate is still dominated by program size. As a result, studying the relationship between inspection fault density and inspection rate is similar to studying the relationship between the size and its reciprocal. That is,

$$\text{relation(fault density, inspection rate)} \approx \text{relation(size, 1/size)}$$

Their Pearson correlation coefficient is usually negative

$$r_p(\text{inspection rate , inspection fault density}) \approx r_p(S , \frac{1}{S}) < 1$$

The empirical data of code inspection fault density and inspection rate is given in Table 3.

Table 3
Code Inspection Fault Density and Code Inspection Rate

Size in NCSL	sample size	inspection rate (NCSL/hour)	fault density (faults/KNCSL)
0 ~ 100	45	60	49
100 ~ 200	36	104	43
200 ~ 400	65	134	33
400 ~ 800	82	153	31
800 ~ 1500	58	171	30
1500 ~ 3000	32	207	26
> 3000	16	297	13

If we focus only on the last two columns of Table 3, we may conclude that higher inspection rate yields lower fault density, and recommend increasing inspection rate to identify more software faults. This seems to be a great finding and no one should oppose such a recommendation. However, such a result is implied by the definitions of these two metrics (rate and density). The recommendation based on these two metrics would not improve product quality or the effectiveness of code inspection. We recommend a general approach with multiple linear regression of code inspection data:

$$\text{inpsection faults} = a + b \times S + c \times \text{inspection time}$$

The best way to eliminate the size factor in this model is to collect enough data points and select only those programs with similar size.

6. CONCLUSIONS

Software density metrics are frequently used by software engineers, especially software fault density. This study identifies several common flaws of applying these metrics in software modeling. Although density metrics are not completely useless, their assumptions are usually not supported by empirical data. Applying density metrics indiscriminately may produce inappropriate improvement plans and waste development resources. Software engineers should check the raw data to verify these assumptions before applying density metrics, and always present density metrics (fault density) along with the original metrics (faults and program size).

7. ACKNOWLEDGEMENT

The original study of density metrics was conducted at Purdue University under the guidance of Prof. H. E. Dunsmore and Prof. V. Y. Shen (currently with the Microelectronic and Computer Corporation). I would like to thank G. D. Huensch and M. J. DiMario for their support of continuing this study. My colleagues, L. A. Bergen, R. W. Bonvallet, M. Gandhi, and W. H. Lin, provided valuable comments on this paper.

REFERENCES

1. F. J. Wall, *Statistical Data Analysis Handbook,* McGraw Hill, Inc., New York, (1986), pp. 1.3 - 1.5.

2. M. Lipow, "Number of Faults per Line of Code," *IEEE Transactions on Software Engineering,* (July 1982), pp. 437-439.

3. N. F. Schneidewind and H. Hoffman, "An Experiment in Software Error Data Collection and Analysis," *IEEE Transactions on Software Engineering,* (May 1979), pp. 276-286.

4. Y. Miyazaki and N. Murakami, "Software Metrics Using Derivation Value," *Proceedings of 9th International Conference on Software Engineering,* (March 1987), pp. 83-91.

5. T. J. Yu, "Software Measurement: Do We Understand What We Are Measuring?" *IEEE Global Telecommunications Conference,* (1986), pp. 1442-1447.

6. V. Y. Shen. T. J. Yu, S. M. Thebaut, and L. R. Paulsen, "Identifying Error-Prone Software – an Empirical Study," *IEEE Transactions of Software Engineering,* 11, 4 (April 1985), pp. 317-324.

7. V. R. Basili and B. T. Perricone, "Software Errors and Complexity: an Empirical Investigation," *Communications of ACM* 27, 1 (January 1984), pp. 42-52.

8. T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering,* (December 1976), pp. 308-320.

9. M. H. Halstead, *Elements of Software Science,* Elsevier North-Holland, New York (1978).

10. J. C. Munson and T. M. Khoshgoftaar, "The Dimensionality of Program Complexity," *Proceedings of 11th International Conference on Software Engineering,* (May 1989), pp. 245-253.

11. D. Potier, J. L. Albin, R. Ferreol, and A. Bilodeau "Experimenting with Computer Software Complexity and Reliability," *Proceedings of the 6th International Conference on Software Engineering,* (1982), pp. 94-103.

12. S. G. Crawford, A. A. McIntocsh, and D. Pregibon, "An Analysis of Static Metrics and Faults in C Software," *The Journal of Systems and Software,* 1 (1985), pp. 37-48.

13. E. J. Weyuker, "Evaluating Software Complexity Measures," *IEEE Transactions on Software Engineering,* (September 1988), pp. 1357-1365.

14. A. S. Wang, *The Estimation of Software Size and Effort: an Approach based on the Evolution of Software Metrics,* Ph.D. Thesis, Department of Computer Science, Purdue University (August 1984).

Software Quality Discrepancy and Code Metrics

Modenna Haney, Regina Palmer and Phil Daley
Martin Marietta Space Systems Company
P. O. Box 179, M/S L4822
Denver, CO 80201

Abstract

Identification of "good" code, i.e., code with a low error rate, through metric scores can free resources to concentrate on finding problems in error prone code, thereby reducing the number of errors found late in a project. This paper presents historical data from a software project on 70 K LOC of "C" code. The coded tools are profiled and their varying error rates discussed. The paper discusses the ease of measuring software quality factors, the relationship between values of these quality factors, the measured incidence of errors and the benefit of using software quality metrics to programmers involved.

Biographical Sketch

Phil Daley is the manager of the Advanced Computing Technology department for Martin Marietta Space Systems Company. His organization is responsible for providing the technology base in advanced software, hardware, and systems for the Space Systems Company. He directs 50 engineers/scientists and three major computer laboratories: Artifical Intelligence, Advanced Spaceborne Processors, and Space Operations Simulation.

Ms. Haney is a senior staff engineer at Martin Marietta Space Systems Company in Software Quality Assurance for the Research and Technology department. She was the assistant program manager of the software project discussed in the paper.

Ms. Palmer is a Software Quality Assurance engineer in the Research and Technology department at Martin Marietta Space Systems Company. She was the SQA lead on the subject program during the last year of its development with oversight duties for system acceptance test.

Software Quality Discrepancy and Code Metrics

Modenna Haney, Regina Palmer and Phil Daley
Martin Marietta Space Systems Company
P. O. Box 179, M/S L4822
Denver, CO 80201

This paper addresses software quality metrics and associated discrepancy data collected on a Research and Technology contract at Martin Marietta Astronautics Group, Space Systems Company. The contract was sponsored by Rome Air Development Center (RADC), Command and Control Directorate, Software Engineering Branch, Griffiss Air Force Base, New York. The original schedule involved 3 incremental software deliveries, 2 hardware deliveries and acceptance test at RADC. The delivered system contained 199K lines of code (LOC) including Commercial Off-the-Shelf (COTS), legacy, and developed software. 38% of the code was COTS or unmodified legacy software. 21% was a conversion of graphics software implemented in VAX Fortran for display on Evans & Sutherland graphics terminals to Fortran 77 on an Apollo DN570 workstation. The remaining 81K was developed under the contract. The manpower on the program fluctuated from 4 to 8 programmers. The software was written in "C", Lisp and Prolog. This paper only addresses data collected for the 70K "C" code.

The development concept for the software was that it be self-documenting. A standardized commentary header was included for each module and was generated automatically. The size goal for modules was 100 LOC or less. The header defined input, output, called and calling routines as well as Program Design Laguage (PDL). The PDL was included in the code to enhance the programmer's ability to maintain and add capabilities during subsequent development.

Quality Factor Measurement

The following 11 quality factors were evaluated for their importance and relevance to the software contract and the five in table 1 were selected for measurement:

correctness	efficiency	flexibility
integrity	interoperability	maintainability
portability	reliability	reuseability
testability	useability	

Portability, reuseability, interoperability, efficiency and integrity were eliminated because of the stand alone nature of the software system being developed. Even though the system was largely defined by its user interfaces, we did not select useability as a factor to measure since the contract did not include measurable standards. Since each quality factor F_Q has measurable attributes x_i, so that $F_Q = F_Q(x_1, x_2, \ldots x_n)$ where the x_i are the criteria by which one may establish the degree of presence or absence of quality factor F_Q the choice of the proper and minimal set x_i} is important and can be established via the concept of "overlap". Overlap occurs when one measurement is indicative of the presence or absence of more than one quality factor. Knowing which criteria overlapped maximally with the Quality Factor allowed us to increase the return on specific measurements. These criteria and their association to Quality factors were selected from seminar publications of Thomas J. McCabe.[3] 12 measurable areas were associated with the five quality factors, as shown in table 1. The areas of most interest to the program management are marked with an " *".

Table 1 Criteria for Measurement and Quality Factors

	Flexibility	Maintainability	Testability	Correctness	Reliability
*Modularity	x	x	x		
Generality	x				
Expandability	x				
*Self-Descriptiveness	x	x	x		
*Consistency		x		x	x
*Simplicity		x	x		x
*Conciseness		x			
Instrumentation			x		
Completeness				x	
Error Tolerance					x
Accuracy					x
*Traceability		x	x	x	

Completeness, even though it had no overlap with the other factors, was used in this study because our standard review process yielded the data. Conciseness was measured because it was possible to automate the collection of the data for the calculation of the observed module length. We did not collect data for instrumentation, expandability, error tolerance or accuracy. Our time and resources were limited and the other factors seemed to be more important. Therefore our selected quality factors, Flexibility, Maintainability, Testability, Correctness, and Reliability were measured by modularity, self-descriptiveness, consistency, simplicity, traceability, completeness, and conciseness. These were measures understood and recognized as meaningful by the group involved.[4]

After the first internal baseline of the code, two samples were established. Sample A represented legacy code and new code produced by programmers with 2 years experience. It contained 154 modules in 7.3K LOC, average size of 47 LOC. Sample B was code from new coders who had been with the company less than 1 year. It contained 176 modules in 5.6K LOC, average size of 32. The sampling included:

1. a one time, manual examination for completeness of the 330 modules representing the total sample against the design described in the PDL. This review is discussed in Attachment I.

2. automated evaluation of all "C" code produced during the 2 year development effort for code simplicity, self-descriptiveness, and conciseness.

A Pascal program was developed on an Apollo workstation to collect data necessary to compute the Halstead Measure [1] used for the conciseness evaluation. It was also used to collect information relating to code simplicity and number of comments. It counted such items as branching, complex logic usage, gotos, lines of code, variable density and non-blank line comments. A requirements traceability matrix was implemented on the VAX to provide a measure of requirement traceability from the statement of work through the requirements and design documents.

Automated Code Review for Sample A and B

The automated evaluation of the code covered items which had no set acceptance level but could yield

data about individual programmer techniques. The following four areas were rated:

1. modular implementation - through measurement of violations of the LOC standard, the value represents 1 - (number of modules that are > 100 LOC) ÷ total number of modules

2. self-descriptiveness - through the number of comments per total lines, the value is (number of comment lines) ÷ (total number of lines in the module).

3. conciseness - Halstead measure by module, the normalized value is 1 - (module length calculated - module length observed) ÷ (module length observed).

4. coding simplicity - through measure of negative Boolean or complicated compound Boolean expressions used (1 - number of expressions ÷ number of executable statements), number of branches (1 - number of branches ÷ number of executable statements), variable density (1 - number of variable uses ÷ maximum possible uses).

Other fixed criteria which had to be met, were evaluated as part of the engineering review process and are not considered in these numbers. These included top down design, hierarchical structure, descriptive mnemonics, parameter agreement between calling and called routines, and use of a structured language.

Table 2 shows the composite scores for the 330 modules in Sample A and B. The comparison becomes more interesting at the end of the project after significant revisions occurred in the code. Sample A had a 24% change rate, code growth of 19% but still improved its score. Sample B experienced a 52% change rate, only 7% growth and a reduction in score. The relative closeness of the system scores gives no clue as to the quality of the code as measured by errors per 1K LOC. The Sample A code had an 8 errors per 1K LOC rate over the entire life of the contract, but Sample B had 19 errors per 1K LOC for the same period.

Table 2 Composite Scores for Sample A and B

	Baseline Sample A	Contract End Sample A	Baseline Sample B	Contract End Sample B
Halstead	.67	.69	.72	.70
% Comments	.52	.52	.58	.59
Variable Density	.47	.48	.42	.42
Branching	.77	.77	.76	.74
Boolean	.96	.96	.94	.91
Modularity	.85	.83	.93	.91
Totals	4.24	4.25	4.35	4.27

Though the code in each sample was generated by teams of programmers, the majority of maintenance on each sample was performed by one programmer. The main programmer of sample A was the most experienced programmer on the program, while the main programmer maintaining sample B was the least

experienced. We believe a correlation exists between maintaining the coding simplicity scores throughout changes in the code and lower discrepancy rates. It is also possible that better programmers demonstrate an ability to maintain consistency, thereby maintaining lower error rates.[5]

Automated Code Review for All Project "C" Code

Table 3 shows data for the rest of the 70K of "C" code developed. The plan for code baseline called for three incremental releases. The first released software (A1 - A5) was in use for 19 months before acceptance testing, the second (B1 - B2) had 14 months and the third (C1 - C4) 6 months. As it turned out, a rewrite was required of two of the tools from the first release just 2 months before the scheduled start of acceptance test (A2 and A5). These were the only tools with a higher incidence of errors during the acceptance test phase than the development phase. A2 had 5K of its 13.4K LOC rewritten but A5 was a total redesign and implementation. The previously discussed Sample A was a subset of Tool A1 and Sample B a subset of tool A4. The table is segmented by baseline with A2 and A5 separated from the first release because of their rework.

Table 3 Tool Data

	1st Release			2nd Release		3rd Release				Re-Release	
	A1	A3	A4	B1	B2	C1	C2	C3	C4	A2	A5
Halstead (old)	.68	.68	.71	.68	.69	.63	.67	.67	.62	.66	.67
(new)	.69	.70	.67	.69	.70	.63	.69	.67	.57	.65	.68
Ave LOC	45	44	38	31	28	31	32	43	100	46	35
	49	46	42	33	29	33	34	45	140	45	35
% Comments	55	58	58	62	56	60	46	56	37	58	58
	56	61	61	62	62	60	56	56	38	59	59
Variable Density	.43	.42	.39	.36	.48	.40	.44	.40	.55	.41	.33
	.45	.41	.41	.34	.47	.42	.46	.40	.62	.40	.34
Branching	.78	.80	.76	.80	.83	.79	.78	.78	.79	.80	.78
	.78	.79	.70	.80	.82	.79	.78	.78	.81	.80	.77
Boolean	.96	.97	.93	.95	.93	.93	.93	.90	.98	.97	.94
	.96	.96	.91	.95	.91	.93	.93	.90	.94	.97	.94
Modularity	.91	.90	.94	.97	.99	.95	.95	.93	.62	.90	.95
	.89	.90	.91	.95	.97	.95	.96	.93	.61	.92	.96
Total LOC (K)	7.9	8.0	5.7	5.6	5.9	6.1	4.8	2.6	1.6	13.5	3.4
	9.1	8.1	6.8	5.7	6.4	6.5	5.3	2.7	3.2	13.4	3.4

Again, the scores for the individual tools do not deviate significantly. C4 and A4 had the lowest system scores and showed the greatest decline in Halstead score. These two also had two of the highest error rates of the tools. System score was not a reliable predictor of performance though, since the two tools with the highest system scores were not the tools with the lowest error rate.

Tool A1 and A4 Profiles

We went back to the software in A1 and A4, as representing the extremes of error rates, and examined the distribution of individual module scores in an attempt to uncover some trait that might be used as a flag during change control or software review board deliberations. Since we had all program error data and all software changes, we decided to look for a link between the original code, one of these automated measures and the known error rate. The average values for the Halstead and LOC per module favored A4, but the real distributions show A1 with fewer low scores in branch and Halstead value as shown in the following three figures.

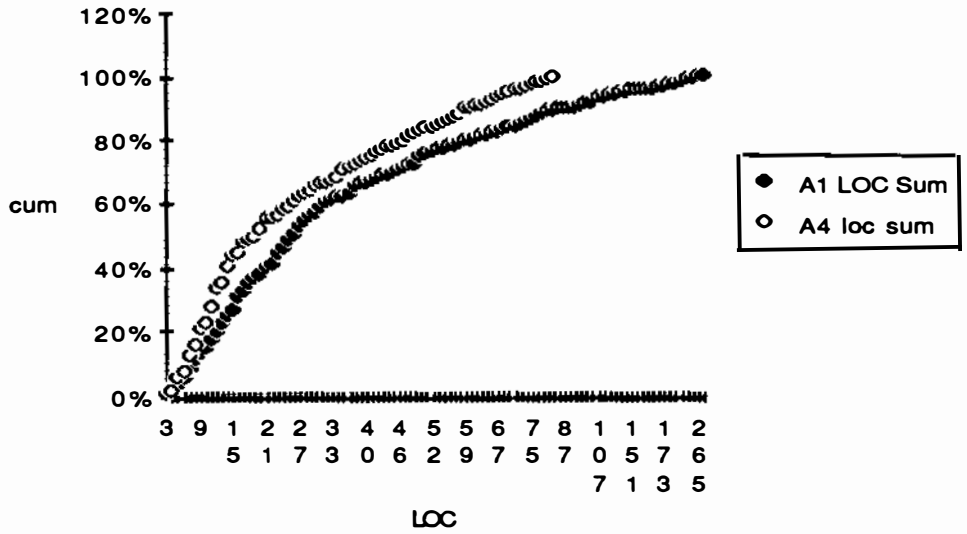


Figure 1 Cumulative Distribution of A1 and A4 Module Size

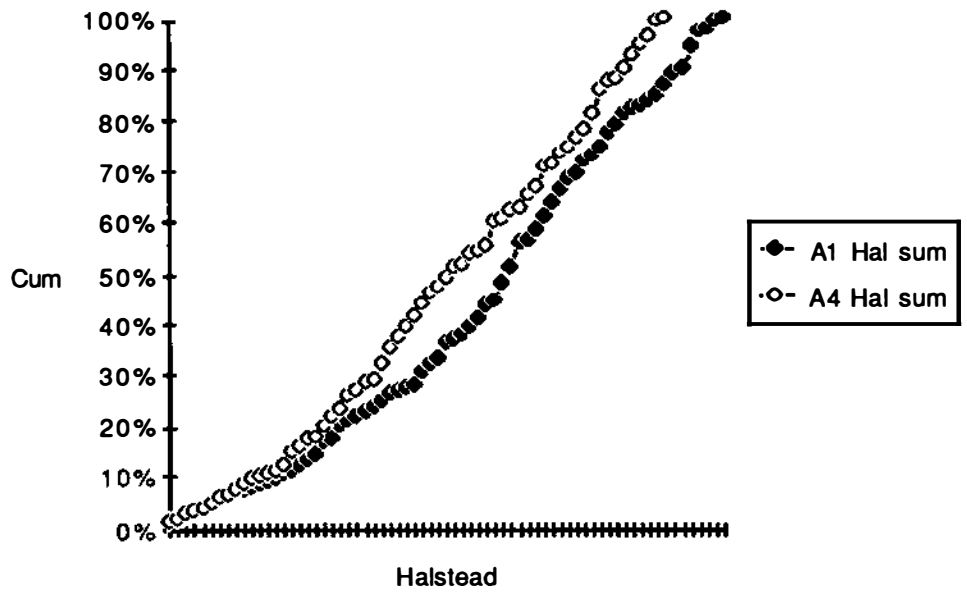


Figure 2 Cumulative Distribution of A1 and A4 Halstead Scores

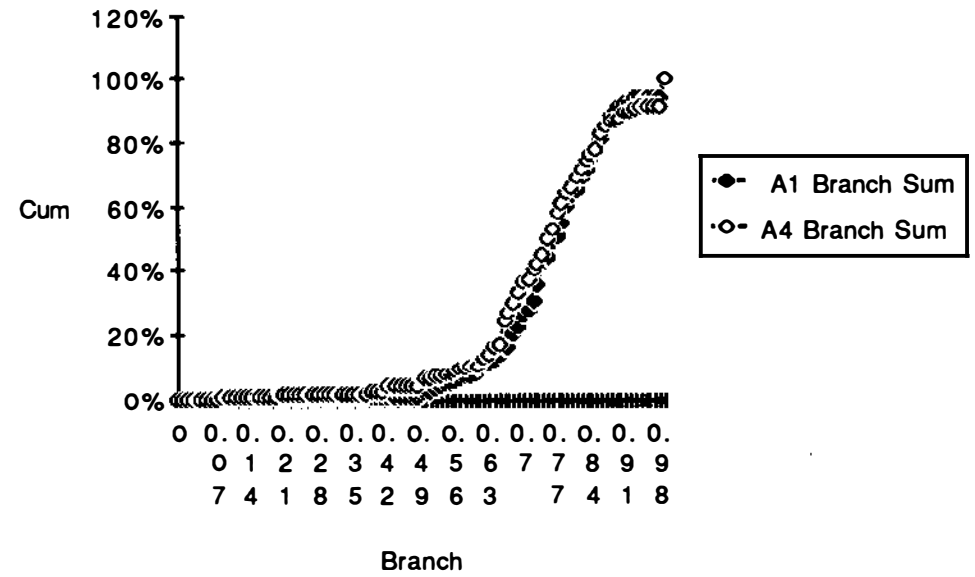


Figure 3 Cumulative Distribution of A1 and A4 Branch Scores

Errors in A1 as a Guide to Errors in Other Tools

We then examined the change traffic in A1 related to coding errors which accounted for the majority of error reports on the tool. Here we found apparent pointers to a threshold for the branch score, Halstead or LOC that yield more incident of error than the general population of modules. The 55 A1 coding errors involved 88 of the 186 routines in A1, representing 5.2K LOC of the final 9.1K LOC.

42 routines of the 60 with LOC > 49, (22% of all modules), were involved in 65% of the errors. The other 35% of errors involved 46 routines of 126 with LOC < 49, (25% of all modules). 51 routines of 106 with branch scores < .80, (27% of modules), figured in 80% of the coding errors. The remaining 20% of errors involved 37 routines of 80 with branch score ≥ .80., (20% of all modules). When examined for Halstead value, 87% of errors involved 62 routines of 137 with a Halstead value < .80, (33% of modules) and the other 7 or 13% involved 26 routines of 59 with Halstead ≥ .80, (14% of modules). Using these as rough guidelines, we calculated an estimated coding error rate for the other tools. The average error rates for the modules in A1 with LOC > 49 versus LOC ≤ 49 are .60 and .15. The average error rates for the modules in A1 with branch score < .80 versus branch score ≥ .80 are .42 and .13. The average error rates for the modules in A1 with Halstead < .80 versus Halstead ≥ .80 are .35 and .12. If A4 could be expected to display a similar pattern of coding errors then A4 with 33 routines with LOC > 49 and 117 with LOC < 49 could be expected to have at least 38 coding errors.

This approach underestimated the coding error rates for 7 of the 10 tools. The exceptions were A5, B1 and C1. Table 4 shows a tabulation of calculated errors if the tools behaved as A1. The numbers in parenthesis signify the percent of underestimation of the known errors. The ratio to A1 error rate does reflect the reality of the project personnels' skill levels as compared to the standard "good" programmer who coded A1.

Table 4 Estimated Errors

	EST ERRORS FROM LOC	EST ERRORS FROM HAL	EST ERRORS FROM BRANCH	ACTUAL ERROR	RATIO TO A1 ERR RATE
A2	44 (.48)	50 (.40)	45 (.46)	84	1.0
A3	40 (.57)	48 (.49)	49 (.48)	94	1.9
A4	38 (.62)	42 (.58)	50 (.49)	99	2.4
A5	17 (.37)	27 (0)	27 (0)	27	1.3
B1	30	47	46	25	.73
B2	27 (.56)	54 (.11)	47 (.23)	61	1.6
C1	30	58	56	48	1.2
C2	27 (.51)	39 (.29)	43 (.22)	55	1.7
C3	16 (.33)	14 (.41)	16 (.33)	24	1.5
C4	9 (.74)	4 (.88)	4 (.88)	34	1.8

Profiles of the Code with Similar Error Rates to A1.

The following 3 charts show the profiles for B1, C1 and A1. B1 which has a lower error rate than A1, in general, has smaller modules with higher Halstead and branch scores.

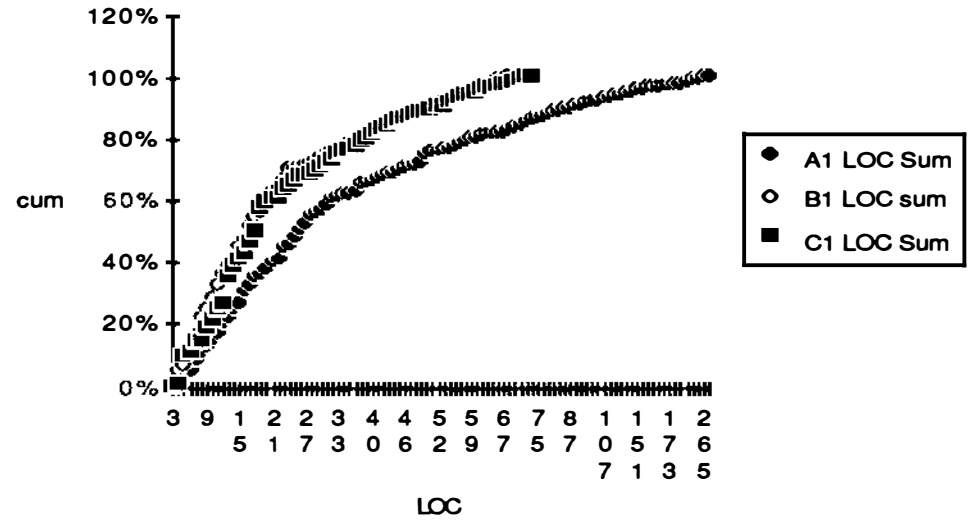


Figure 4 Distribution of Module LOC Size in A1, B1, and C1

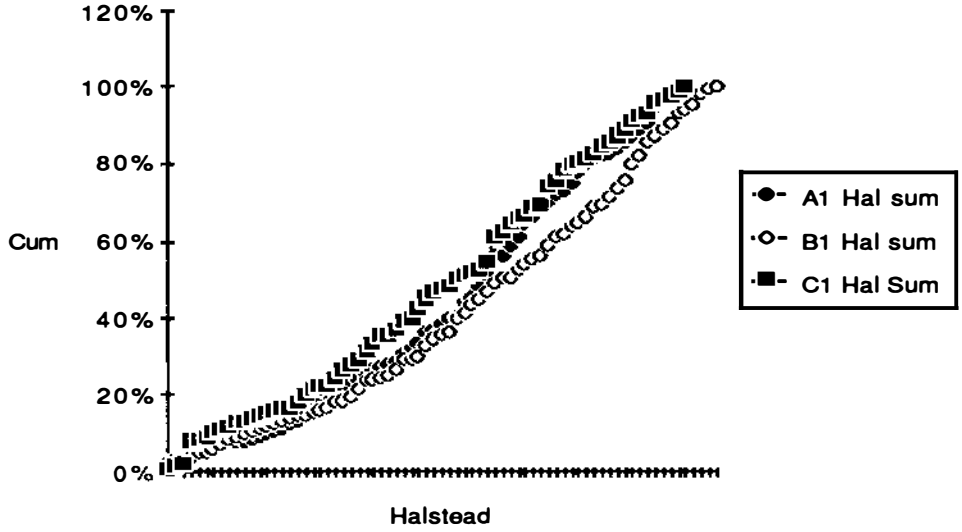


Figure 5 Distribution of Halstead Score in A1, B1, and C1

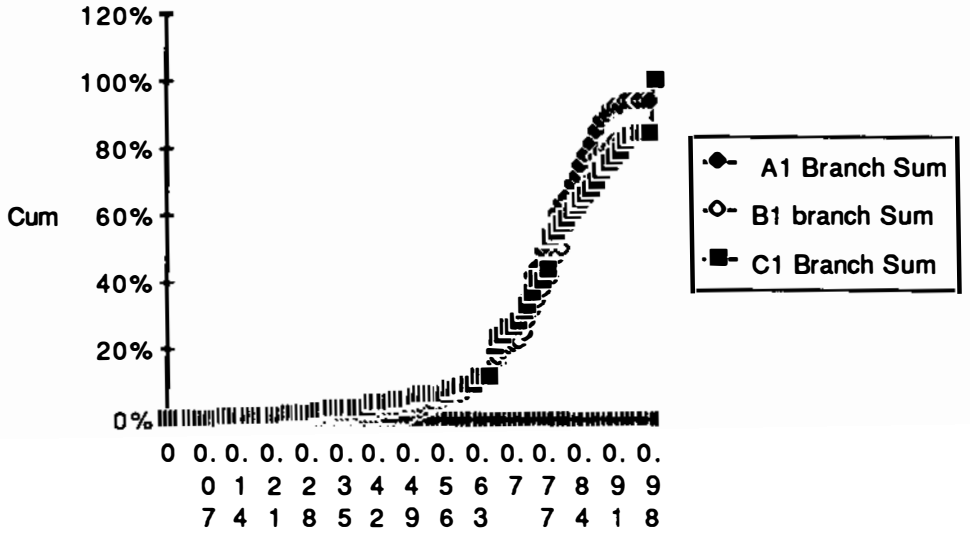


Figure 6 Distribution of Branch Scores in A1, B1, and C1

This appears to underline the relative merits of the individual programmers involved, at least as compared to the expected errors based on an acknowledged 'good' programmer. In dealing with any of these measures the quality of the programmer stills seems to be the determinate factor.

Changes to Metrics as the Software Evolved

As noted before, system scores for each tool did not yield a satisfactory identification of the good versus the bad. A pattern does seems to emerge when the change within each tool from the baseline to the end of contract is examined as in Table 5. The tools generally had a main programmer responsible for incorporating changes. The programmers have been designated in the Table as P1 through P5. After parts of it were rewritten, a main programmer could not be determined for tool A2. The tools are grouped by baseline release with the exception of A5 and A2, the two tools that experienced rewrites at the end of the program. The programmers showed varying inclinations toward complexity as changes were introduced to the original baselines. The definition of errors and their distribution in the tools are discussed in Attachment 2.

Table 5 Changes Within Each Tool

TOOL	ERRORS/ 1KLOC	TOOL SIZE(K)	MOD. AVE LOC	AVE. BRANCH SCORE	Δ BRANCH	AVE. BOOLEAN SCORE	Δ BOOLEAN	Δ B&B	SYSTEM SCORE	Δ SCORE	HAL-STEAD	ΔH
A4 (P5)	19	6.8	42	.70	-.06	.91	-.02	-.08	4.21	-.10	.67	-.04
A3 (P2)	17	8.1	46	.79	-.01	.97	-.01	-.02	4.37	+.02	.70	+.02
A1 (P1)	8	9.1	49	.78	0.00	.96	0.00	0.00	4.33	+.02	.69	+.01
B2 (P3)	13	6.4	29	.82	-.01	.91	-.02	-.03	4.49	+.01	.70	+.01
B1 (P4)	8	5.7	33	.80	0.00	.95	0.00	0.00	4.35	-.03	.69	+.01
C4 (P5)	15	3.2	100	.81	+.02	.94	-.04	-.02	3.93	0.00	.57	-.05
C2 (P3)	12	5.3	34	.78	0.00	.93	0.00	0.00	4.38	+.15	.69	+.02
C3 (P4)	11	2.7	45	.78	0.00	.90	0.00	0.00	4.24	0.00	.67	0.00
C1 (P3)	9	6.5	33	.79	0.00	.93	0.00	0.00	4.28	+.02	.63	0.00
A5 (P3)	8	3.4	35	.77	-.01	.94	0.00	-.01	4.32	+.03	.68	+.01
A2	8	13.4	45	.80	0.00	.97	0.00	0.00	4.33	+.01	.65	-.01

The tools with the greatest negative change in the boolean and branching values, A4, C4, A3 and B2 have a higher error rate than those with no change, A1, A2, B1, C1, C2, and C3. The boolean and branching values are a reverse measure, i.e., the higher the score the lower the incidence of either in the code measured. Amongst the tools with no change, the error rate was lower for those with higher combined Branching and Boolean scores with the branching value apparently having the greater influence. [2]

To investigate this further, we examined the values for the individual files which received the most changes with the 2 best and 2 worst error rates, A4, A3, A1 and A2. The first table (Table 6) shows values for tool A4 which had the worst error rate. The individual routines, F1 - F7 are numbered according to

original size. The programmer P3 did 37% of the changes in the files shown in table 5 and P5 did 63%. Programmer P5 was responsible for the changes in the first 4 files, had no input in the fifth file and split the updating job with P3 on F7. The large positive change in the branching and boolean numbers for F7 was mostly contributed by P3.

Table 6 A4 Values

A4	REVS	HAL-STEAD	Δ HAL-STEAD	BRANCH SCORE	ΔBRANCH	BOOLEAN	ΔB OOLEAN	ΔB&B	% GROWTH
F1	22	.30 .48	-.18	.65 .70	-.05	.86 .98	-.12	-.17	85
F3	10	.59 .56	+.03	.77 .82	-.05	.97 .99	-.02	-.07	-16
F2	6	.42 .49	-.07	.72 .80	-.08	.97 .99	-.02	-.10	01
F4	6	.96 .92	+.04	.75 .74	+.01	.92 .92	0.00	+.01	06
F5	2	.53 .45	+.08	.75 .70	+.05	1.0 1.0	0.00	+.05	-02
F7	2	.51 .64	-.13	.78 .67	+.11	.78 .67	+.11	+.22	33
F6	1	.15 .30	-.15	1.0 1.0	0.00	1.0 1.0	0.00	0.00	25

Tool A3 (Table 7) had changes made by four programmers. The confusion in the numbers may be indicative of the programmer egos as each programmer attempted to change the style of the previous programmer. Programmer (P2) was the primary creator of A3 and did the majority of changes in the three files with the most revision levels, these three files appear to follow the pattern recognized in A4. P2 did 50% or fewer of the changes in the remaining 4 files. The two largest files with the fewest changes were added to A3 by P1 in response to a requirement change and were mostly maintained by P1.

Table 7 A3 Values

A3	REVS	HAL-STEAD	Δ HAL-STEAD	BRANCH SCORE	ΔBRANCH	BOOLEAN	ΔB OOLEAN	ΔB&B	% GROWTH
F4	10	.96 .93	+.03	.72 .80	-.04	.95 .98	-.03	-.07	-01
F3	9	.75 .72	+.03	.72 .76	-.08	.92 .91	+.01	-.07	30
F5	7	.73 .63	+.10	.90 .91	-.01	1.0 1.0	0.00	-.01	-34
F6	6	.85 .74	+.11	.74 .87	-.13	.96 .97	-.01	-.14	-12
F7	6	.76 .76	0.00	.79 .79	0.00	.71 .71	0.00	0.00	0
F2	5	.71 .82	-.09	.75 .82	-.07	.98 .97	-.01	-.08	19
F1	4	.76 .74	+.02	.72 .74	-.02	.95 .95	0.00	-.02	08

When one of the tools with the least rate of errors is examined, it also seems to indicate that no change is superior to either positive or negative change in the branching and boolean scores. This tool (Table 8) was almost completely maintained by programmer P1.

Table 8 A1 Values

TOOL A1	REVS	HAL-STEAD	ΔH	BRANCH SCORE	Δ BRANCH	BOOLEAN	Δ BOOLEAN	Δ B&B	% GROWTH
F1	9	.92	+.14	.78	+.03	0.98	0.00	+.03	+16
F4	8	.80	-.15	.77	-.03	0.96	+.02	-.01	+102
F7	7	.65	-.01	.80	+.07	1.00	0.00	+.07	+06
F3	4	.73	-.03	.79	+.04	1.00	0.00	+.04	00
F10	4	.52	-.02	.86	+.01	1.00	0.00	+.01	+12
F2	3	1.0	+.02	.83	0.00	0.99	0.00	0.00	-04
F9	3	.90	-.02	.80	0.00	0.93	0.00	0.00	+05
F5	2	.69	+.04	.80	0.00	1.00	0.00	0.00	+03
F6	1	.58	-.04	.86	0.00	1.00	0.00	0.00	00
F8	1	.92	+.03	.74	0.00	0.85	0.00	0.00	-02

When the routines in tool A2 (Table 9) were examined, they also seem to indicate that smaller swings in the metrics translate to fewer additional revisions.

Table 9 A2 Values

A2	REVS	HAL-STEAD	Δ HAL-STEAD	BRANCH SCORE	ΔBRANCH	BOOLEAN	ΔB OOLEAN	ΔB&B	% GROWTH
F4	10	.07	-.22	.75	+.11	.95	+.02	+.13	+42
F1	8	.85	+.08	.68	0.00	.93	+.02	+.02	+35
F2	6	.96	+.15	.67	+.03	.97	-.01	+.02	-18
F3	1	.67	-.01	.65	+.02	.97	0.00	+.02	+3

The examination of these numbers can not be divorced from the programmers involved. Beginning scores do not give a good indication of lurking coding problems if those scores have little variance. Deterioration of these scores though may offer a flag to change inspectors as to whether additional problems are being introduced.

The observables of the code, such as the metrics used here, do not readily translate into identifying which code is likely to cause heavy change traffic. But as the code is tested and changed, the tendency of the metrics to change can help flag problem areas. Tools A1 and A4 when studied show divergence in their

rate of error within 3 months of the development baseline as shown in Figure 8.

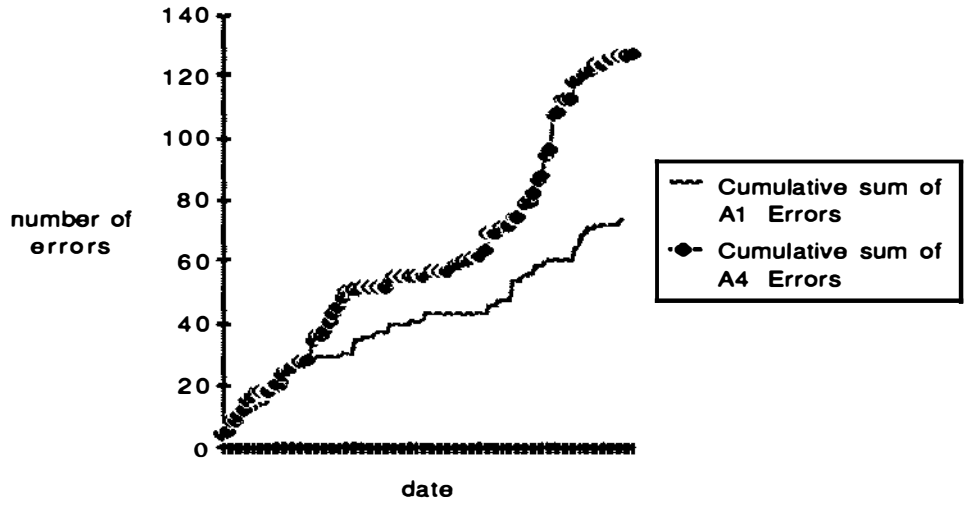


Figure 8 Divergence of Errors

Figure 9 shows the effect change had on the branch scores for the first 25 changes in each tool. The steadier state shown by A1, we believe indicates better code or programmer discipline.

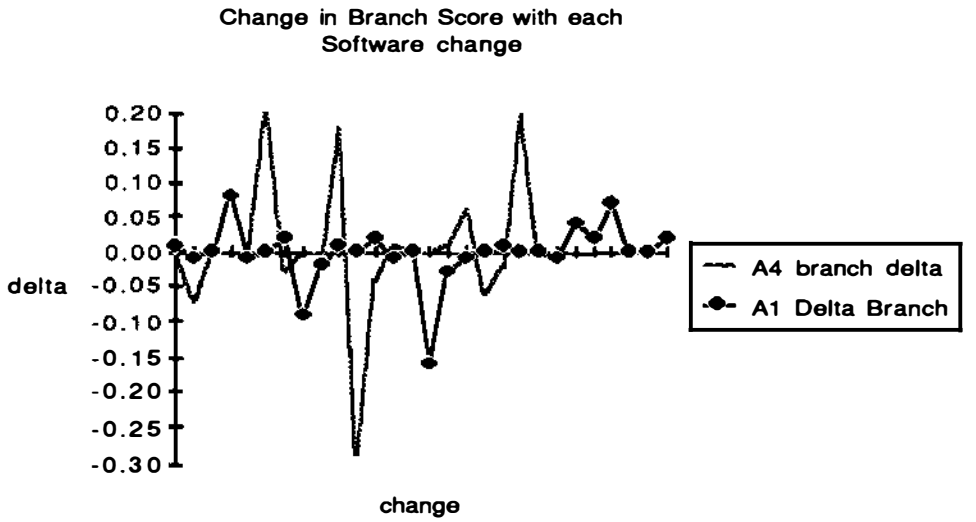


Figure 9 Branch Score Changes

Figures 10 and 11 show the engineering hours related to fixes in tools.

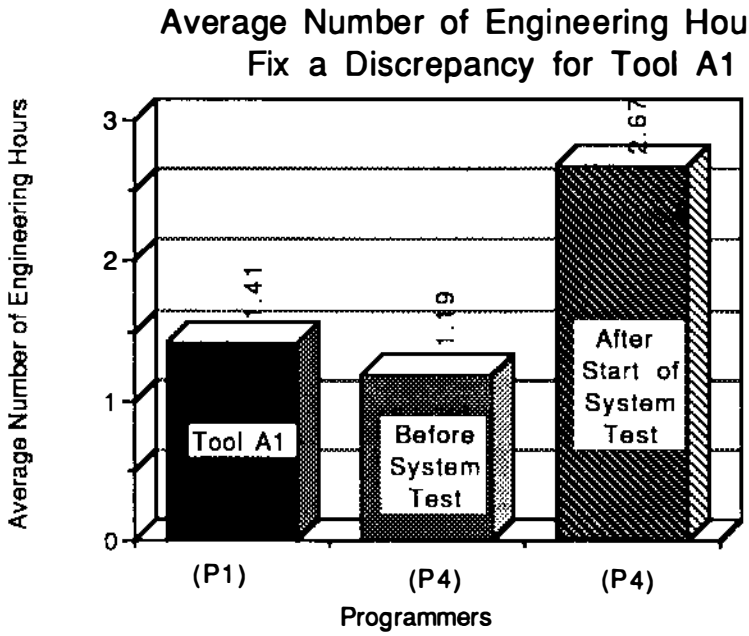


Figure 10 Average Number of Engineering Hours per Fix

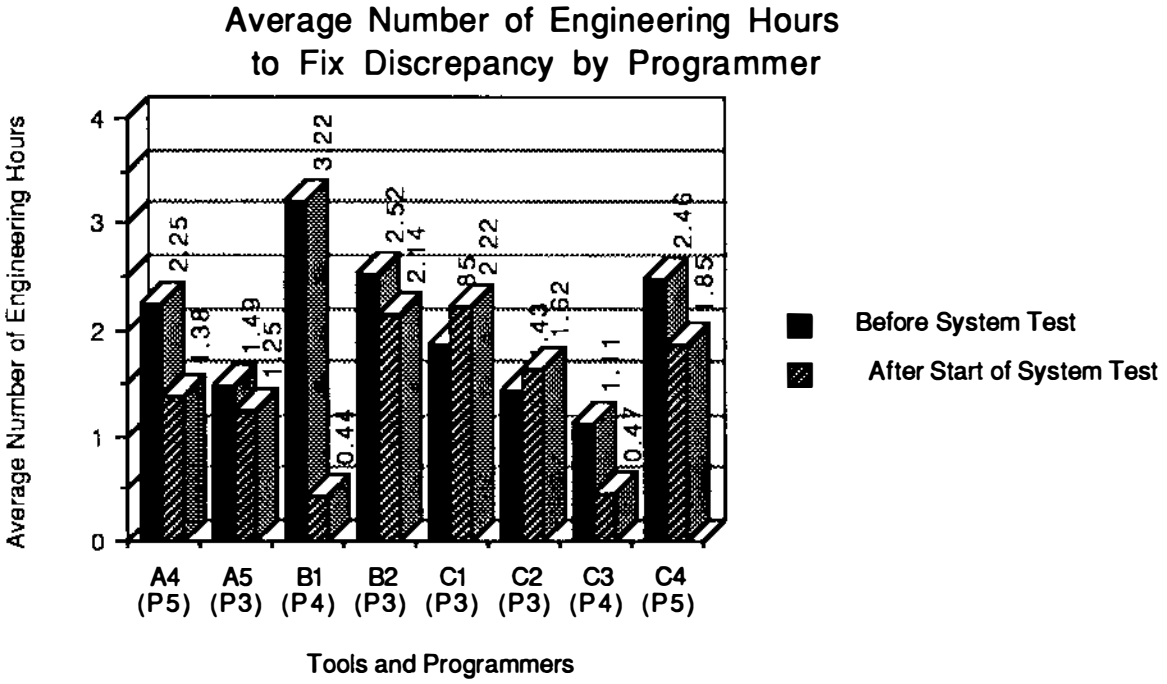


Figure 11 Average Number of Engineering Hours per Fix

Results of Quality Measurements

Our data shows that following the changes in the McCabe branch measure as code is revised gives valuable insight to programmer habits that can effect the quality of code. Programmers who maintain lower

error rates in the code they are responsible for do not, on average, introduce changes in metric values as discrepancies are worked.

Lower error rates are found in tools with module distributions skewed toward simplicity and conciseness as measured by branch score and Halstead scores. But the risk of high error rates for modules with low scores or even large module sizes are reduced by programmer selection. Better programmers can work against the odds involved in changing error prone code. The profiles of the quality measures by module reflect the differences in the programmers in relation to their experience level. On project it was noted that retest skills improved with time, so that, resubmittals of corrections by the better programmers were more likely to be correct the first time.

The experience gained with this data has prepared us to embark on a more active use of metrics to support change control evaluations. The majority of time spent by software quality assurance after the initial baseline of this code was spent in assuring the configuration of the code for subsequent test activities. The project was different from most previously encountered efforts since the initial baseline that quality controlled was a true development baseline not a system test baseline. Regardless of the initial quality of code or time in the development cycle, reviewing code changes for proper incorporation of a change needs to be supplemented with re-evaluation of the "quality" of the code after a change. Having a tool to automatically flag changes in conciseness or decision points can be used to help assure that the code is not degraded by change and that test case changes are not overlooked.

A side effect of this study is related to the large reuse of code on this program through incorporation into new routines. Though a coding error might be found in one version of a reused module, we had no automatic configuration management system that could ensure that the original or all permutations of it were also corrected. And similarly, the testing tied to the reused code could not be back propagated. We have not had time to evaluate if there was a significant incidence of errors amongst reused code, but recognize that it could become a problem on future programs if adequate configuration management systems are not available.

References

1. M. Halstead, *Elements of Software Science*. New York: Elsevier, 1977.
2. R.K. Lind and K. Vairavan, "An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort", IEEE Trans. Software Engineering, Vol. 15, SE 5, May 1989.
3. T. J. McCabe, "Software Quality Assurance: A Survey", Conference Notes 1979.
4. J. H. Poore, "Derivation of Local Software Quality Metrics (Software Quality Circles)", Software-Practice and Experience, Vol 18(11), November 1988.
5. M. Takahashi and Y. Kamagachi, "An Emperical Study of a Model for Program Error Prediction", IEEE Trans. Software Eng., Vol 15, SE 1, January 1989.

Attachment I

Manual Review

The manual review of the sample was performed to examine each module for code completeness. The criteria were understandable PDL, internal consistency between the PDL and code, adherence to contract coding standards, and self-descriptiveness. A score of 0 to 1 in half point increments, with an additional weighting point given to internal consistency, was assigned to each criterion for a 5 point maximum. The review was performed by non-developmental personnel from Software Quality Assurance (SQA). The following figure shows the distribution of scores for the samples.

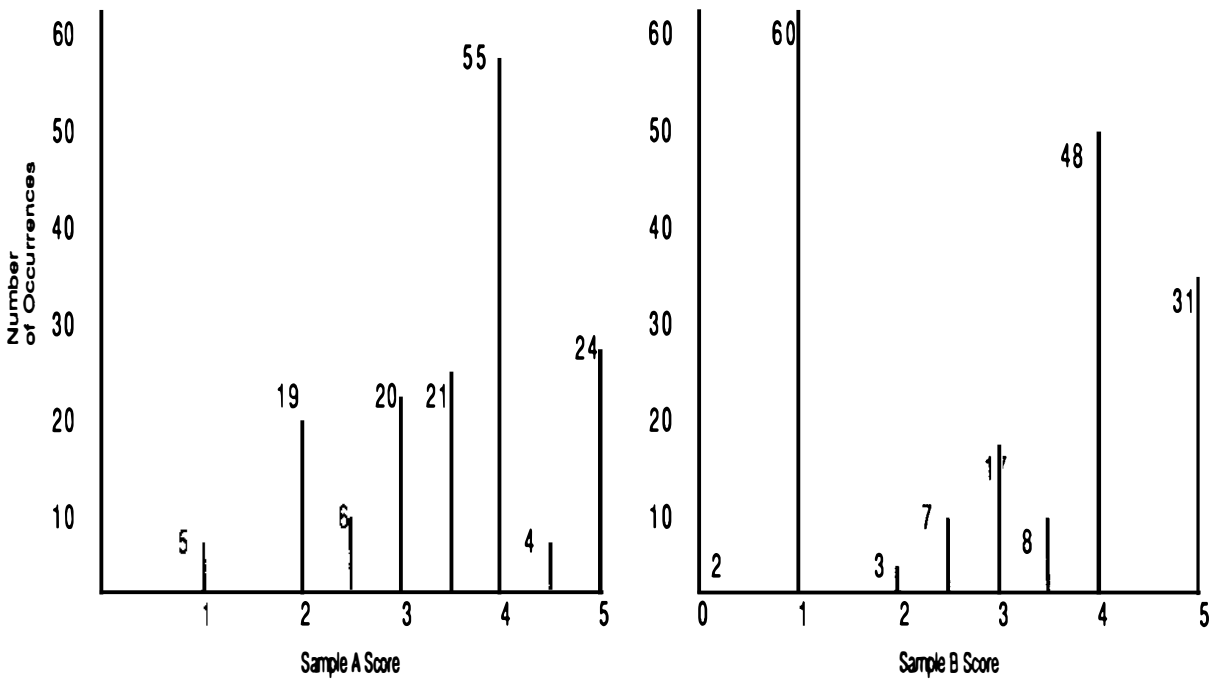


Figure 1 PDL to Code Consistency Scores

81 % of Sample A modules were above average, only 59% in Sample B. The acceptance rate of sample A was 97%, sample B 65%. The mean score in sample A was 3.6 while the sample B mean was 2.9. Within sample A the probability that a module would be excellent was 54% versus 45% for sample B. Sample B ratings tended to be either very good or very bad with only 20% falling in the middle. The results of the sampling were distributed to the coders and a limited time for rework effort authorized. The programmers of sample A chose to rework only the 11 worst cases increasing their acceptance rate to 100% with the mean score rising to 3.8, but the members of sample B reworked 70 modules, changing the average score of these 70 from 1.1 to 4.9, and raising the mean score of sample B to 4.4. The rework covered 23% of the modules, resulted in an 100% acceptance rate and raised the total sampling mean score from 3.2 (good) to 4.1 (excellent).

Attachment 2

Determining Discrepancy Data

The process of tracking discrepancies of software provides information to other programs and to management to help improve productivity and efficiency. The information compiled here compares the number of discrepancies found prior to system test and the number of discrepancies found after system test. It also includes the severity of the errors found in these phases.

There were 913 discrepancies found during the software life cycle phases. There were 626 discrepancies found prior to the start of system test and 287 discrepancies were found after the start of system test. Of the 626 discrepancies found prior to the start of system test, 81% were found by the contractor and 19% by the customer. Of the 117 discrepancies found by the customer, 19 were considered enhancements by the contractor. These 19 discrepancies have been taken out of the figures for the total number of discrepancies found prior to start of system test. Of the 287 discrepancies found after the start of system test, 97% were found by the contractor and 3% by the customer. Of the 287 discrepancies found after the start of system tet, 2 were enhancements and have been taken out of the figures for the total number of discrepancies. The system acceptance test at the customer facility consisted of a three week effort in which 35 discrepancies were discovered. Of the 35 discrepancies, 7 were documentation updates, 2 were non reproducible, 1 required no action, 4 were installation errors and 14 required code changes. 69% of the total discrepancies were found prior to system test and 31% after the start of system test. For the entire project, 4.0 discrepancies per 1000 lines of code were found during system and integration testing. The error rate during the entire development and test phase was 12.7 per 1000 lines of code.

There are 5 severity levels for discrepancies found by the contractor or customer. These discrepancies range from "A" to "E". An "A" discrepancy is an error in the code in which the software does not meet the requirements or design, an error which is a documentation error which causes the code to not meet the requirements or a code error which crashes the system making the system nonoperational until the error is fixed. A "B" discrepancy is an error which crashes the system but there is a work around and the system can be used. A "C" discrepancy is an error found in the code which does not interface with the operation of the system. A "D" discrepancy is a minor error in the code such as a typographical error in a help message. An "E" discrepancy is an enhancement to the current system, directed by the contractor or the customer. Table 1 identifies the number of discrepancies by severity level before system test and after the start of system test.

Table 1 Severity of Discrepancies

	Before System Test			After Start of System Test		
Severity	Number of Discrepancies	Average Number of Engineering Hours to Fix Discrepancy	Average Number of Lines of Code Changed Per Discrepancy	Number of Discrepancies	Average Number of Engineering Hours to Fix Discrepancy	Average Number of Lines of Code Changed Per Discrepancy
"A"	28	2.68	69	9	2.23	247
"B"	56	6.67	59	18	2.80	128
"C"	185	3.54	46	133	1.96	52
"D"	288	2.95	73	121	1.69	58
"E"	50	2.36	70	4	1.14	23

The table also shows the average engineering hours (Figure 1) and average number of lines of code changed per discrepancy before system test and after the start of system test. As the programmers became more experienced, the average number of engineering hours to fix a discrepancy improved for all severity levels of discrepancies.

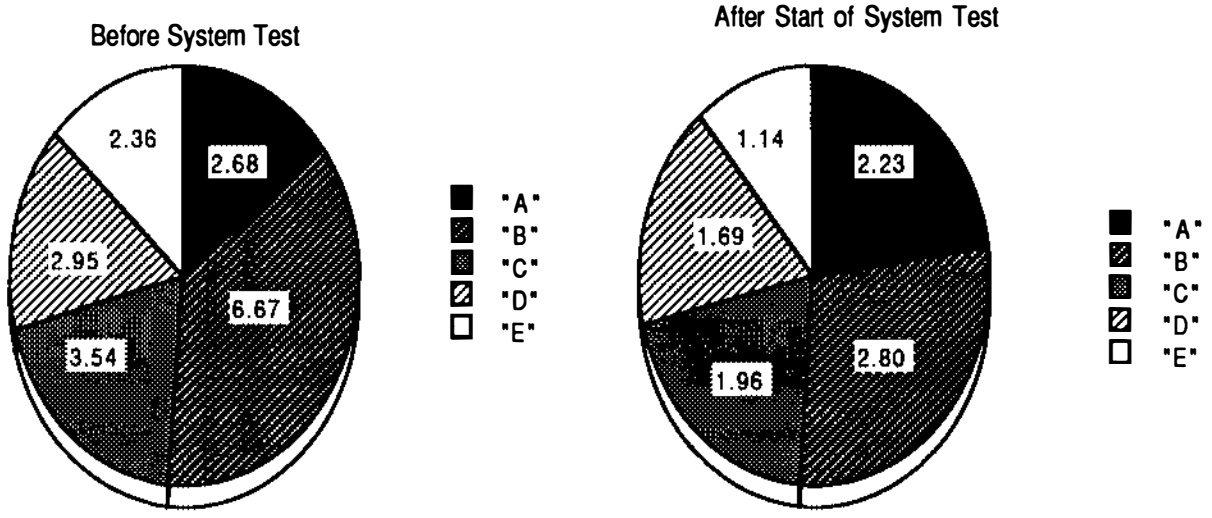


Figure 1 Average Number of Engineering Hours to Fix a Discrepancy by Severity

Figure 2 shows the distribution of errors by severity. Prior to the start of system test, 5% "A" discrepancies, 9% "B" discrepancies, 30% "C" discrepancies, 47% "D" discrepancies and 8% "E" discrepancies were documented. Of the 28 "A" discrepancies found, 6 discrepancies were for software documentation that did not meet the contractual requirements. The number of "A" discrepancies and "B" discrepancies found before system test indicates that the code was not tested sufficiently prior to baselining the code. After the start of system test, 3% "A" discrepancies, 6% "B"

discrepancies, 46% "C" discrepancies, 42% "D" discrepancies and 1% "E" discrepancies were found. Of the 9 "A" discrepancies found, 5 were for software documentation that did not meet the contractual requirements. After the start of system test, one of the "A" discrepancies was a system crash and the others were requirements and design errors. The number of "B" discrepancies found during system test indicate insufficient time for the engineer to test the code in sufficient detail before turning over modified code.

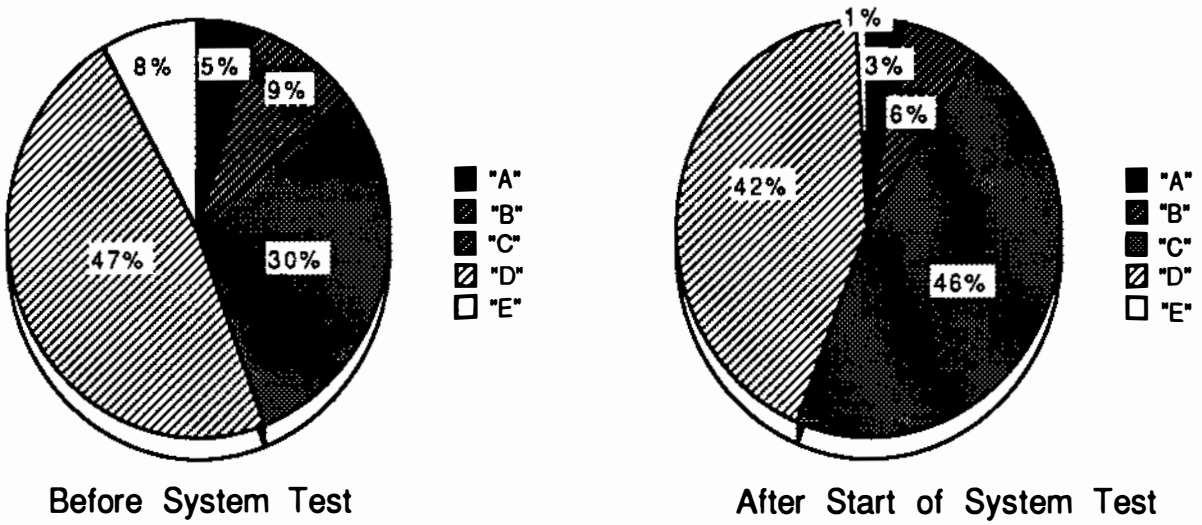


Figure 2 Distribution of Errors by Severity

The following table presents the data from the 11 tools which comprised this system. It identifies each tool, the number of discrepancies against each tool, the discrepancies per 1000 lines of code, the total number of lines of code changed and percentage of lines of code changed, and the % of total errors found before system test and after the start of system test.

Table 2 Total Discrepancies and Lines of Code

	Before System Test					After Start of System Test				
Description	Number of Discrepancies	% Errors Discovered	Discrepancies Per 1000 LOC	Total LOC Changed	% LOC Change	Number of Discrepancies	% Errors Discovered	Discrepancies Per 1000 LOC	Total LOC Changed	% LOC Change
A1	60	81%	6.6	3706	41%	14	19%	1.5	689	8%
A2	51	48%	2.7	3826	29%	55	52%	2.9	5714	43%
A3	100	72%	12.5	6782	84%	38	28%	4.8	3530	44%
A4	99	78%	14.6	6371	94%	28	22%	4.1	1007	15%
A5	12	43%	3.5	425	13%	16	57%	4.7	632	19%
B1	34	77%	6.0	3189	56%	10	23%	1.8	460	8%
B2	66	80%	10.3	7886	123%	17	20%	2.7	1286	20%
C1	35	58%	5.4	2320	36%	25	42%	3.8	918	14%
C2	38	61%	7.2	896	17%	24	39%	4.5	954	18%
C3	15	52%	4.2	714	26%	14	48%	3.9	86	3%
C4	32	68%	10.0	959	30%	15	32%	4.7	2769	87%

A1, A3, and A4 were improved between the end of development test and the end of system test because of the increased experience level of the programmers. A3 increased in size from the original baseline because of the addition of increased functionality. A4 increased in size when error checking was added.

A2's increase in discrepancies and percentage of lines of code changed was caused by the tool being rewritten two months into system test. System test lasted three months. After the baseline, 29 discrepancies were written against the new A2. In the first two months of system test, the old A2 was tested. The new A2 had 3.2 discrepancies per 1000 lines of code and a 22% line of code change after the new baseline. System test for A2 also represented development test.

There were several types of errors found during testing. There were Coding Errors, Design Errors and Requirements Errors. Coding Errors represented 74% of all errors, Design 16% and Requirements 2%. Even after the start of sytem test, 2% of the errors were in design or requirements. The ratio of errors found during the testing phases is shown in Figure 3.

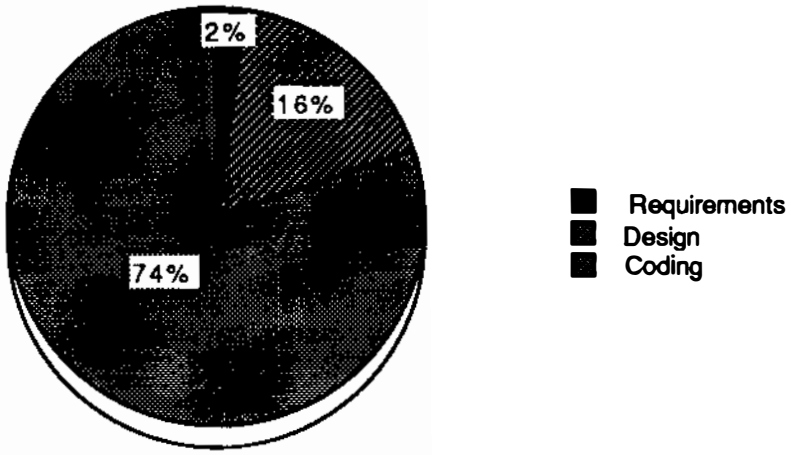


Figure 3 Ratio of Types of Errors Found During Testing

After the end of the coding phase and before development testing, the majority of the requirements and design errors should have been detected and fixed. The PDR and CDR packages did not contain a visible design presentation which might have lent to the large number of requirements and design errors that went undetected through the coding phase. The project was on a tight schedule to deliver prototypes of each of the tools which interfered with review time. Table 3 identifies the types of discrepancies that were found before and after the start of system test.

Table 3 Types of Discrepancies

	Before System Test			After Start of System Test		
Description	Number of Requirements Errors	Number of Design Errors	Number of Coding Errors	Number of Requirements Errors	Number of Design Errors	Number of Coding Errors
A1	4	14	40	0	1	13
A2	0	13	22	3	1	52
A3	0	27	59	2	1	35
A4	1	20	75	0	4	24
A5	0	1	11	0	0	16
B1	1	12	17	1	1	8
B2	3	15	44	0	0	17
C1	1	8	23	0	0	25
C2	1	2	33	1	1	22
C3	0	3	10	0	0	14
C4	0	8	21	0	2	13

Several different reasons figured into the number of requirements and design errors found during development test and system test. The requirements definition was insuffient for A1 prior to baseline. The requirements and design for A2 were not fully implemented by the original engineer. The large number of design errors after baselining shows a poor implementation of the documented design. It was due to these errors discovered after start of system test that A2 was rewritten by a different programmer so close to the end of the program. We surmise that the number of design errors for A3 and A4 were caused by the immaturity of the documentation at implementation time. This was caused partially by the nature of the project which emphasized the prototyping of interfaces before the production of documentation.

Towards an Object-Oriented Analysis Technique

Dennis de Champeaux
Walter Olthoff

Hewlett Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
(415)-857-6674 champeaux@hplabs.hp.com
(415)-857-7877 olthoff@hplabs.hp.com

Keywords: Object-Oriented Analysis, Design Methodologies, Formal Software Development

Abstract

This work is motivated by the mismatch between Structured Analysis (SA) and object oriented design for implementations in object-oriented programming languages (e.g. data dictionaries provide little guidance for classes and their hierarchal structure). We explore the replacement of SA by a method based on the object-oriented paradigm. Extensions and substantial modifications of the method outlined in Shlaer and Mellor's book [SM88] are described. The extensions include additions to the *information model* and *state model*. We introduce an additional model, the *interface model*. We sketch our process model that differs substantially from the process models in [SM88].

Biographical information:

Dennis de Champeaux: Senior scientist at HP Labs since 1986; about 25 publications on: heuristic search, theorem proving, verification, medical data bases, etc. Current interests: object-oriented analysis, integrated software development environments and tools, (parallel) algorithm and code validation.

Walter Olthoff: Development engineer at HP Labs since 1987. Received Dipl.-Inform. degree from University of Bonn, FRG, in 1981; received Ph.D. in computer science from University of Kaiserslautern, FRG, in 1987. Worked in formal software specification methods, program verification and object-oriented language design. Current interests: development of formalisms for system analysis, especially in combination with object-oriented paradigms.

Towards an Object-Oriented Analysis Technique

Dennis de Champeaux
Walter Olthoff

1. Introduction

Object-oriented analysis (OOA), like structured analysis (SA), aims at clarifying an initially informal description of a software development task. OOA differs from SA in that it acknowledges that an implementation is done in an object-oriented programming language. Identification of objects, which extend data dictionary elements in SA, gets a higher priority in OOA and is done before the behavior is analyzed. We may summarize the difference between SA and OOA as: entities are *inferred* from behavior in SA, while in OOA behavior is *superimposed* on entities. This difference allows the postponement of addressing the temporal dimension of the system until one has obtained detailed descriptions - in static terms - of the entities in the domain of interest.

The commonality between SA and OOA is that their output is a description of what the task is about and not of how the task is to be solved. Consequently, it is essential that the language used in the analysis phase is user oriented; the user should be able to recognize both what has been communicated to the analyst and what deviations from an intended description have crept in. As a corollary the description language cannot cater for implementation idiosyncracies. For instance, no description should be biased towards a sequential versus a parallel architecture.

Usage of a graphic language is common in analysis, although English (or the reader's preferred language) still plays an important role. Our work aims at strengthening the expressiveness of a graphic formalism, reducing the role of English and thus paving the way for formal interfaces between the phases in the development process. However, there are certain aspects of the analysis activity that we have not yet resolved For example, performance requirements are outside our notation. Similarly, resource restrictions cannot be handled. Thus we have so far concentrated on the characterization of functional and behavioral aspects.

This paper motivates and elaborates the following extensions of the method outlines in Shlaer and Mellor's book [SM88]:

- The analysis phase does not necessarily start with a requirements document in natural language. Instead, we offer semi-formal notations to support development of such a document.

- Information modeling is enriched with apparatus to describe attributes beyond naming them, while expressing constraints on attributes.
- With regard to state modeling: for a transition to fire, an arc not only prescribes satisfaction of a condition but also demands the reception of a trigger.
- The notion of an interface model that "projects" out the state from the state model. Whether this "transition" view of an object *precedes* the construction of the state model or alternatively *follows* it is as yet under investigation.
- Our process model addresses concurrent phenomena and makes explicit inter-object interaction.

We expect these additions to give improved guidance for the subsequent design phase. Moreover, our emphasis on formalisms makes validations of the target system against its requirements easier. Relying on formalisms also contributes to a common language between customer and analyst.

A follow up goal for this work is to architect a tool for OOA. The method we are developing has to be precise so that a supporting tool could formalize the output, allowing tight integration with other tools that support the design, implementation and debugging phases.

Section 2 describes in some detail our present version of OOA. We sketch what we took over from [SM88] and describe some major extensions. Section 3 deals with an application of our method to a classic example, the car cruise control system. We summarize our work and give some conclusions in section 4.

2. Presentation of OOA

2.1 Our Heritage from Shlaer/ Mellor

Following Shlaer and Mellor [SM88], we identify the following three models generated during the analysis phase:

- information model
- state model
- process model

Another borrowing from [SM88] is their use of the word *object* (which entails what is usually understood as *class*). An object is an abstraction of a set of real-world things such that:

(1) all of the real-world things in the set - the instances - have the same characteristic;
(2) all instances are subject to and conform to the same rules.

An *instance* of an object is an entity that has the characteristics required by the object.

The information model describes the relevant objects in the domain of interest, while ignoring the temporal dimension. An internal view is described by salient attributes and through an external view by recording invariant relationships among objects. Exploiting conceptual hierarchies helps conciseness and avoids redundant descriptions.

The state model describes dynamic behavior of each object separately through state-transition diagrams, but it leaves out object interaction. The states in the model should correspond on a one-to-one basis with externally recognizable states of the object.

The process model finally describes the "social" behavior of an object. Actions associated with states are introduced, which may refer to attributes, states or transitions of other objects. The causal behavior is captured as well.

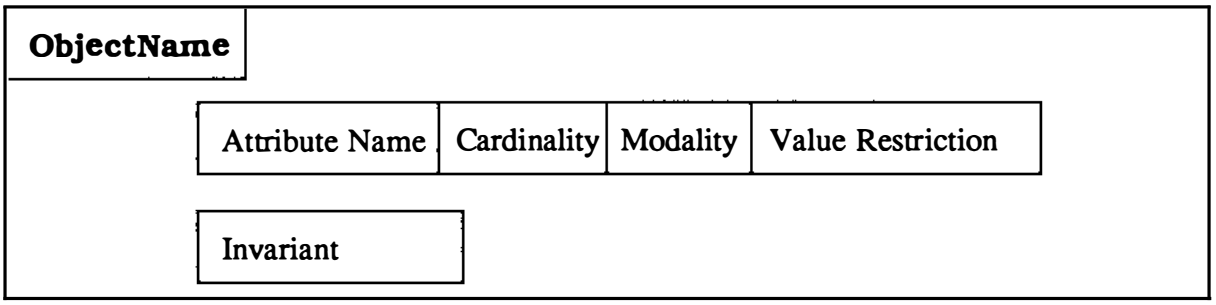
2.2 Extensions

Our experience with applying Shlaer and Mellor's method has led us to the conclusion that there are two major deficiencies: (1) the expressiveness and formality is limited that hampers not only what can be handed over to the design phase but also restricts the role for generating the benchmarks for the target system; (2) inter-object connections (sometimes called object interfaces) are expressed only indirectly through the vehicle of "classical" dataflow modeling of the actions that are associated with the states of an object. We have developed extensions that address these issues. In section 3. we give examples of the concepts introduced below.

2.2.1 Information Modeling

The information model (IM) is based on finding objects and their attributes (we disregard the process of finding here, but note that [SM88] contains a comprehensive list of guidelines). The analyst in Shlaer and Mellor's version describes an attribute by simply creating a descriptive name (like *BankAccount*). Even though it is often obvious that a particular set of numbers denotes the values of *BankAccount*, this is not expressed in the IM. Also, attribute values in actual instances might be supplied by default, or only explicitly; for the second case, the analyst lacks a way to express it. Third, to express conditions that have to be valid for all instances of an object (called *invariants*), the IM provides only textual and unstructured comments.

To allow greater sophistication in expressing attribute features, we extend the notation for modeling objects by the following:



The following information can be attached to an object and its attributes:

- Attribute Name: a locally unique name.

- Cardinality: whether the attribute value is a singleton, enumeration, fixed or unbounded subset.
- Modality: whether this attribute always has a value (i.e. is mandatory), optional or whether this attribute is derived.
- Value Restriction: the named set of values out of which actual values have to be taken.
- Invariant: allows the expression of a condition that holds for all instances. It is a truth-valued expression that refers to features of attributes. An object can have more than one invariant.

Note that attributes are not required to have all this information; if it doesn't, we write "?" as a placeholder. But the analyst should provide it, if possible.

Objects defined in this style can share attributes. In that case, the common attributes can be abstracted into a new object, and subtyping (c.f. [SM88]) relates them to the reduced objects.

2.2.2 State modeling

For each object identified in the IM of [SM88], the state model includes the following information:

- a set of states, where a state is defined by analyzing the intended object behavior; attached to states are actions that are pseudo-code descriptions of effects of entering the state;
- a set of transitions that describe how an instance of the object changes its present state to a new state; transitions happen as the result of events.

We put stronger emphasis on the state model because it provides the initial formal description of the object's behavior. States are derived from the set of all possible attribute values of all attributes. No attributes of other objects are taken into account. Usually, a state is characterized by a condition on certain attributes while ignoring other attributes. The initial and terminal state of an object receive special treatment. Both are depicted as arrows into (out of) state vertices where the source (target) is missing.

We do not associate actions with states, but with transitions. This modification allows different actions to be done when a state is entered dependent on where the transition originated. Since events are caused by actions in [SM88], they have a different way of modeling causal consequences. Their model can be phrased as "states cause each other", while our method captures "transitions cause each other".

We augment the concept of state transitions by attaching

- a *condition* that is to be fulfilled before a transition can take place. That is, being in a state does not automatically enable a transition. Such a condition can refer to attributes of "other" objects;
- an *external flag* that indicates that an external event is required;
- a *cause list* that explicitly contains events that are caused as a consequence of the transition (for the notion of event, see section 2.2.3).

The augmentation with conditions and external flags gives more expressivity to the analyst; the cause list solves an important deficiency we experienced in applying [SM88]. They have

only a rudimentary way of connecting state machines, namely by indexing events in data flow diagrams associated to process models. This indirection hides the interaction between objects, and later it makes program design more difficult. We need to express explicitly that a particular transition in one state machine causes a particular transition in another. For example, in a model of an accelerating car with objects Tire and Engine with corresponding state machines, the event of pressing the accelerator causes two state transitions :

Engine{slow_rpm → high_rpm}, and
Tire{no_rotation → rotation}

2.2.3 Process Modeling

[SM88] introduces data flow diagrams (DFD's) from structural analysis to describe the actions associated with transitions. DFD's are their process model, but it does not express directly the causal relations between objects. The interrelationships are buried inside DFD's and show up as commonly used external data stores. Data flow diagrams are Shlaer and Mellor's glue for different state transition diagrams.

As mentioned, we have introduced a direct connection between distinct state diagrams by associating cause lists with transitions. We define *events*, the elements of these cause lists, as follows:

> An event is a set of transitions of the form *obj.trans*, meaning that the transition *trans* has to take place in the state machine for the object instance *obj*. There is at most one occurrence of *obj* in a particular event.

An event bundles and synchronizes transitions of multiple objects. Objects that participate in an event can freely exchange data; i.e. descriptions of to-states can rely on features of from-states from all participating objects.

An abstracted version of *message passing* is used in OOA. Since objects are autonomous and concurrent entities, message passing in the form of procedure calls is too restrictive. We have chosen an asynchronous form of inter-object interaction that factors out the data-transfer aspect of interaction. What remains is only the triggering aspect. Triggers are used for capturing causal connections among objects in the domain of discourse.

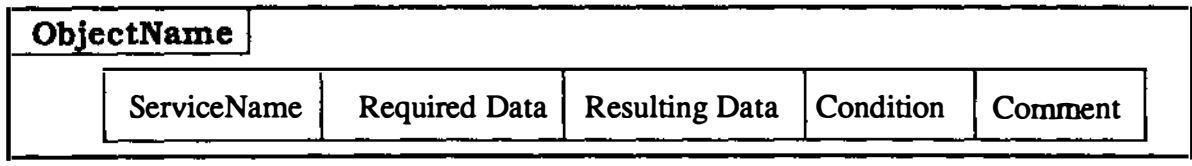
2.2.4 Interface Modeling

Our notion of an interface model is tentative and still under development. At issue is whether "services that an object provides" need to be or can be described *independently* of the collection of states that one has identified for an object.

Up to now, the object model used in OOA has comprised attributes and behavior, as given by the state transitions and their attached action, condition, trigger information, and cause list. In several cases we used that information to describe other transitions in other state models ("in object *obj*, transition *trans* occurs" (c.f. events)). By doing so, we have indirectly decided that particular transitions are visible in the global scope (especially all those transitions,

whose *external* flag is true). We will call such transitions a *service* of an object. Once a state and process model are completed, one can derive for a given object a set of such transitions.

Another option is to define beforehand for a given object what its services are, and then use this new requirement in state and process modeling. We call the set of services provided by an object its *interface*, and the process of determining it, *interface modeling*. An interface model for an object may include the following information:



Each provided service has a name. If available, the analyst can also provide information of what particular data is required to perform that service, and what the resulting data is. Also a condition can be given that governs whether a service request is handled, or how the object attributes are effected by performing that service (using pre- and postconditions). An interface model describes object behavior by services that are performed by the object; it summarizes what clients have to be able to "ask" the object.

The two options of deriving the interface model (from state and process model, or by requiring it) still have to be investigated for the respective advantages and disadvantages. Independently, modeling object interfaces explicitly during the analysis phase is an addition to object-oriented analysis, because it prescribes what *forms of usage* can be made of an object. Also, the interface model is a formalization for measuring the output of later development phases whether they cater for what was identified in OOA.

There is no a priori assumption that services correspond to transitions one-to-one. In the first cut of the analysis often services and transitions are the same. But if the state machine associated with an object is refined during the analysis process, new states and new transitions are introduced. The salient point with object interfaces is that they are stable under such refinements. As a consequence, eventually a service can correspond to a sequence of transitions.

In the interface of an object, we can distinguish two kinds of "services":

- one in which the object only reports about its private state of affairs; and
- one in which the object performs some activity, that changes its state and that is best seen as being performed independent from the service requesting object.

Reporting services can be derived easily from the conditions in those transitions that refer to non-local attributes. The activity services will be discussed below in more detail. We have not looked yet at the issue of creation and deletion services.

In the previous sections, we looked at the progression:

1) describe an object using attributes, inter-object relations and intra-object constraints;
2) identify public states of an object;
3) identify transitions that connect states;
4) describe the actions associated with a transition by detailing how the to-state relates to the from-state in the transition; and

6) describe the "social behavior" of objects, i.e. indicate how a transition triggers subsequent transitions.

This progression is sensitive to the set of states, which is determined by specific features of the application domain. Obtaining services (and associated actions) independent from the application domain can be done

- by abstraction, using the transitions developed on top of the identified set of states. Preparing the transfer from the analysis phase to the design phase is the primary advantage of the interface model in this alternative.
- by exploiting intuitions that an analyst has in an earlier phase regarding the actions that an object should be capable to support. The interface model plays a more prominent role in shaping the understanding of the domain of discourse in this approach.

In comparing interface modeling with [SM88], we note that only few aspects of "how is an object used" can be captured by *relationships* and *associative objects* (i.e. relationships with additional structure). These notions originate in data base modeling and theory where there is clear distinction of what an object is and where it is used. Therefore relationships are decoupled from individual objects and live in an orthogonal domain; also, objects are not constrained to occur only in a defined set of relationships. It has to be proven, whether this application of data base technology concepts to object-oriented analysis is really appropriate.

Actual practice should determine the proper role of the interface model, and in fact, we are prepared for the possibility that there is no definite answer. Certain tasks may promote one approach while other tasks may benefit from the other.

3. The Automobile Cruise Control Example

We will analyze in an object-oriented fashion the Automobile Cruise Control example. The full version of the analysis can be found in [CO89].

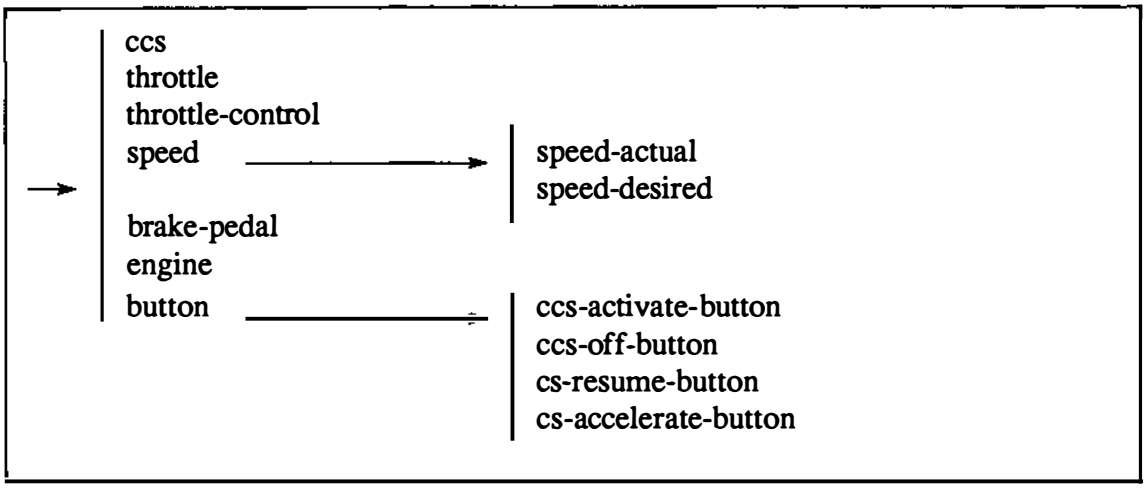
The analysis originates with an idea usually described in a somewhat vague and incomplete document. As an example we used the following text, which was taken from [WPM89]:

> In normal operation, the driver of an automobile monitors the actual speed of the vehicle by watching the speedometer and moving the accelerator pedal to keep actual speed close to desired speed. The cruise control system relieves the driver of this responsibility by automatically maintaining the speed requested.
>
> The cruise control system can operate only when the driver has started the engine successfully. When the driver activates the system, the system stores the current speed as the desired cruising speed and maintains that speed by monitoring actual speed, computing the desired throttle position, and setting the throttle actuator to that value. While cruising, the driver can request the system to gradually accelerate, then stop acceleration and use the current speed as the new desired cruising speed; this is done with two buttons on the cruise control panel. Pressing the brake pedal stops the acceleration and suspends cruising; pressing the "Resume" button on the cruise control panel resumes cruising.
>
> The driver may deactivate the system at any time by pressing the "Off" button on the cruise control panel. Turning off the engine also deactivates the cruise control; the hardware implements this by initiating a "Deactivate" event when the engine is turned off. If the driver wants to temporarily slow down, he or she can press the brake pedal; the driver can then instruct the system to resume cruising by pressing the "Resume" button.

From it we selected the following entities to be taken into account in the analysis:

ccs (= cruise-control-system)
throttle-control
brake-pedal
speed-actual
ccs-activate-button
ccs-resume-button
ccs-stop-accelerate-button
throttle
engine
speed-desired
ccs-off-button
ccs-accelerate-button

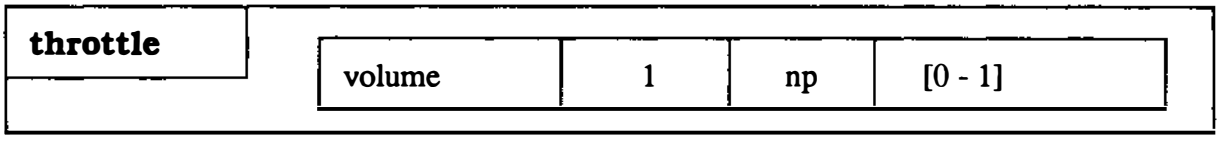
This list describes the entities constituting a cruise control system. Instead of describing instances, we describe the class to which it belongs. To make class descriptions minimal, we look for classes that have something in common to push the shared features into a superclass. The buttons and speeds are candidates for abstraction, and so we obtain the hierarchy:



3.1 Information Model

In the information model phase, we describe each object internally through attributes and if necessary with constraints on attribute combinations, and externally through relations among objects that cannot be captured through attributes.

A typical example of an object is given below; it will be extended later.



Volume is an attribute name; the cardinality *1* indicates that there is only one volume atribute for each throttle; the modality *np* (for "necessarily-present") indicates that this attribute must occur in every instance; the value restriction *[0 - 1]* requires that actual values of volume are inside that interval.

We intend to model the throttle as an autonomous "machine" that constantly adjusts the volume. *Throttle* has three tight "loops". The loop to be executed depends on the value of the *mode* attribute in *throttle-control* that can be *manual*, *cruising* or *accelerating*. In *cruising* mode, the new *volume* is determined by the *volume* of the current state, by the *desired speed* and by the *actual speed* (see below).

Similarly, we have:

throttle-control				
	mode	1	np	[manual,cruising,accelerating]

The value-restriction is an enumeration of three distinct values.

3.2 State Transition Model

The state transition models are to be made for each entity introduced in the information model. We decompose this activity in two steps:

- identify the set of states;
- give the transitions that connect states.

In theory, one should be able to formulate a characterizing predicate for each state. Since an object can be only in one state at the time, the conjunction of any number of different characterizing predicates is false.

Although we introduce redundancy, it is a good practice to adjoin a special attribute to a object that keeps track of the state. For *throttle*, which has only one state, we obtain:

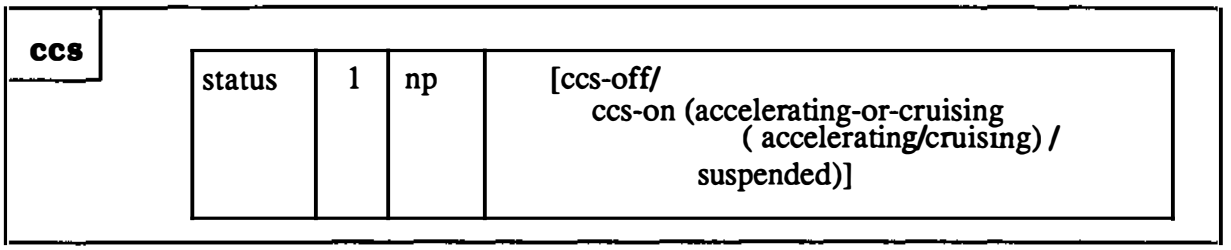
throttle				
	volume	1	np	[0 - 1]
	status	1	np	[state-of-throttle]

A more interesting state case is *ccs* :

ccs				
	status	1	np	[ccs-off, accelerating, cruising, suspended]

Ccs has four different states: *ccs-off*, *cruising*, *accelerating* and *suspended*. The investigation of the transitions for these four states showed many similarities. For instance, there are

identical transitions out of the non-*ccs-off* states to the *ccs-off* state (always the driver can turn off cruise control). The result is the following set of abstracted states for *ccs*:



Thus we have abstracted the states *cruising* and *accelerating* into *cruising-or-accelerating*, while adding *suspended* gives the state *ccs-on*.

After identifying all the states for all the entities we introduce the transitions. *Throttle* has three transitions coming out of and going into the same state:

ccs-adjust-volume
    state-of-throttle ⟶ state-of-throttle
manual-adjust-volume
    state-of-throttle ⟶ state-of-throttle
accelerate-adjust-volume
    state-of-throttle ⟶ state-of-throttle

Similarly, we obtain for *ccs* the following set of transitions:

ccs-turn-on
    ccs-off ⟶ cruising
ccs-turn-off
    ccs-on ⟶ ccs-off
ccs-brake
    cruising-or-accelerating ⟶ suspended
ccs-accelerate
    cruising ⟶ accelerating
ccs-stop-accelerate
    accelerating ⟶ cruising
ccs-resume
    suspended ⟶ cruising

By using the state model hierarchy, we get the transition *ccs-turn-off* to apply, through inheritance, on *cruising*, on *accelerating* as well as on *suspended*.

Naming the transitions is the minimum required for their characterization. However, more detail can be provided. Especially the effect of a transition on attribute values can be expressed with pre- or post-conditions, in a specific *effect* clause. For instance, we can pro-

vide the semantics of the *ccs-adjust-volume* transition inside *throttle* with the following (see [CO89] for more explanations):

throttle

ccs-adjust-volume: state-of-throttle ⟶ state-of-throttle

extraneous-condition:	the-throttle-control.mode = cruising
trigger-required?:	no
causal-consequence:	nil
effect: volume =	compute-volume(self.volume, self.the-speed-actual.speed self.the-speed-desired.speed)

Comments:

- in the *extraneous-condition* a reference is made to an attribute in another object;
- there is no external trigger required for this transition: satisfaction of the *extraneous-condition* is the only requirement;
- there is no *causal-consequence*: no other transition is triggered;
- the *effect* description describes the new *volume* as a function of the old *volume* and two other parameters, which are references to attribute values in other objects.

We find a transition that requires a trigger and has a causal consequence inside ccs:

ccs

ccs-turn-on: ccs-on ⟶ cruising

extraneous-condition:	self.the-speed-actual.speed > min-speed-constant
trigger-required?:	yes
causal-consequence:	set-speed-desired-button

Comments:

- this transition is triggered as a consequence of a transition inside an instance of *ccs-activate-button*;
- to make this transition more realistic, we insist that the actual speed is more than a certain minimum speed, as expressed in *extraneous-condition*;
- the *causal-consequence* refers to *set-speed-desired-button*, which is an indirection towards an event descriptor that "fires up" two transitions in parallel.

3.3 Process Model

The process model is where we deviate most from what is presented in [SM88]. They associate an action with a state. Thus in their conception an action is executed when a state is entered. They describe actions with data flow diagrams.

In contrast, we conceive actions to be executed during arc traversal and we describe these actions solely via post-conditions, either inside the details of a transition or inside the details of an event descriptor. Recall that an event - in our setting - is a set of synchronized transitions involving multiple objects. Our process model consists simply of these event descriptions.

For instance, we mentioned above, that the trigger *set-speed-desired-button* referred to an event description. The latter is:

ccs

```
set-speed-desired-event
      constituent: [  trans set-speed-desired of ccs.the-speed-desired,
                      trans switch-to-ccs of ccs.the-throttle-control]
      effect: let  speed            = ccs.the-speed-actual.speed
                   desired-speed    = ccs.the-speed-desired.speed
                   mode             = ccs.the-throttle-control.mode   in

                   desired-speed = speed
                   mode = cruising
```

Comments:

- the *constituent* field describes two objects and their transitions that are triggered: the current speed becomes the desired cruise speed and throttle control changes into *cruising*;
- the *effect* description captures the semantic of the event: the post conditions of all the transitions that participate in the event.

What to do when an object receives a trigger that it cannot honor (either because an extraneous condition is not satisfied and/or because the object is not in the from-state of the triggered transition) is an open issue. We have the following options:

- such an occurrence is an error;
- the trigger will be buffered;
- the trigger is ignored.

A preliminary investigation suggests that there is no single right answer. Sometimes, we may prefer one of these behaviors, sometimes another. This suggests that we have to look for a better primitive than triggers.

3.4 Interface Model

The interface model describes the services each object provides, independently of how they relate to state transitions or extraction of state information. As mentioned earlier, there is an initial similarity between services and transitions, but it disappears during the refinement of the state model of an object. A part of the interface model for ccs can be described in the following way:

ccs

service	Required Data	Result Data	Condition	Comment
accelerate	s:speed		min< s <max	the-current-speed = s
current-speed		speed		observes only

In this case, we require that the *accelerate* service is supplied with a target speed within a boundary; the comment indicates that the desired speed is eventually reached. The *current-speed* service gives back a speed value; it only observes the state of a *ccs* object. A complete interface model for *ccs* could include services like *enable*, *disable*, *suspend*, *cruise*, etc. Note that the analyst is not required to give information for other than the first column; but if the information is available, OOA provides the formalisms for it.

We developed our model of the cruise control system without initially developing interface models for the entities produced by the information model. Alternatively, we could have conjectured in an earlier phase that the above mentioned services are required, and then confine the description of the state model accordingly. Similarly, we could have identified in an earlier phase the operations that correspond with the transitions in *ccs*. We need to work on more examples to assess the contribution of an interface model to an earlier phase of OOA.

4. Summary

The analysis method described here has been influenced by Shlaer and Mellor's book. By applying what they suggest, we developed several changes that enhance the expressivity available to the analyst and make our method of system analysis object-oriented. Some of the modifications are:

- Attributes are not simply names that denote values out of a domain defined elsewhere. OOA provides for attribute name, cardinality, modality, and value restriction; and this information is optionally incorporated in the object definition.
- Objects have invariants to express requirements that must be satisfied by all instances of the object.

- State modeling is further developed in OOA and carries more information than in Shlaer and Mellor's book. We also address the issue of finding states, and develop a notion of refinement of state machines.
- In the state model description, OOA provides means to express connections between distinct state machines, capturing causal connections, through events that are attached to single state transitions.
- Transitions in state diagrams are not simply source-target arrows, but associate action, triggers and conditions with transitions.
- Actions are connected to transitions, not to states.
- Interface modeling adds the externally accessible services to the description of objects. The analyst may provide information about required data, resulting data, and acceptance conditions for each service. The interface of an object remains unchanged under refinements of the state model.

Although we dealt only with a toy example, the car cruise control system, the analysis activity did reveal an incompleteness in the English language characterization of what such a system is supposed to do. It is unspecified what happens when the cruise control system is in the accelerating mode, the brake is hit and subsequently the resume button is pushed.

We have suggested above that the generation of models happens sequentially. This may be a wise strategy indeed when one does not have machine support for the development of these models. We foresee, however, a tool assisting the OOA activity in which the order of developing the models is not fixed and accommodates ideas as they "bubble up" during the analysis process. Furthermore, such a tool would be integrated with similar tools for the design phase, the implementation phase and the maintenance phase. Such an integration would attack the discrepancies between what the code does, what the design promises and what the analysis prescribes.

Acknowledgements

We are grateful to the people at HP for sharing their experience with object-oriented analysis. Especially Theresa Novak and Will Nicholl spent considerable time in clarifying issues and they helped - and are still helping - to debug our ideas with their real-world experience. Alex Stepanov's help in clarifying the presentation of the material is deeply appreciated. Alan Snyder and George Woodmansee and an unknown reviewer gave valuable comments on earlier drafts. We, of course, are responsible for all remaining errors.

References

[Bai88] Sidney C.Bailin. An Object-Oriented Specification Method for ADA. In Fifth Washington ADA Symposium, 1988.

[Boo88] Grady Booch. On the Concepts of Object-Oriented Design. Unpublished article, Rational Inc., Denver, 1988. (to appear).

[CO89] Dennis de Champeaux, Walter Olthoff. Object-Oriented Analysis Method. Hewlett-Packard Labs, Palo Alto, 1989 (in preparation).

[SM88] Sally Shlaer, Steve Mellor. Object-Oriented Systems Analysis, Yourdon Press, 1988.

[SS86] Ed Seidwitz, Mike Stark. Towards a General Object-Oriented Software Development Methodology. In Proc. First International Conference on ADA Programming Language Applications for the NASA Space Station, pages D.4.6.1 - D.4.6.14, 1986.

[War89] Paul T. Ward. How to Integrate Object Orientation with Structured Analysis and Design. IEEE Software, Vol. 6(2), pp. 74 - 82, March 1989.

[WPM89] Anthony I. Wasserman, Peter Pircher, Robert Muller. An Object-Oriented Structured Design Method for Code Generation. Software Engineering Notes, Vol. 14(1), pp. 32-55, January 1989.

An Example of Large Scale Random Testing

Printer controller software was evaluated using random testing. This method provided an effective alternative to functional testing with advantages for finding program faults and for project management.

Ross Taylor
Tektronix, Inc
P.O. Box 1000, M/S 63-356
Wilsonville, OR 97070-1000
(503) 685-3586
rosst@pogo.WV.TEK.COM

Testing, Random Testing, Test Coverage

Ross Taylor received an A.B. degree in Economics and an M.B.A. degree in Accounting from the University of California in 1975 and 1980 respectively. He has worked as a management information consultant and a financial systems analyst and is certified in Production and Inventory Control at the Fellow level. Presently, he is a software engineer for the Graphics Printing and Imaging Division of Tektronix, Inc., working in the area of software quality assurance.

An Example of Large Scale Random Testing

Ross Taylor
Tektronix, Inc
P.O. Box 1000, M/S 63-356
Wilsonville, OR 97070-1000
(503) 685-3586
rosst@pogo.WV.TEK.COM

1. Introduction -- Two Approaches to Testing

Software testing methods can be divided into two fundamental approaches, random testing and partition testing. At present, partition methods are more commonly used than random methods, partly due to their intuitive appeal and partly due to difficulties with random testing in many applications. Research comparing random and partition testing has found that the appeal of partition testing may not be justified.[1] [2] This paper describes an application where large scale random testing was feasable and advantageous. Since some partition testing was also carried out, the two methods could be compared in terms of the faults that each method detected. Random testing was found to be quite effective for this application.

1.1 Random Testing

Random testing consists of selecting test cases at random from the input space of the program. The program is applied to these test cases and the results are compared to results that have been determined to be correct. Ideally, the probability distribution of the selected inputs should be the same as the probability distribution of the inputs that the program will encounter in actual use (the operational distribution.) This is particularly true if the percentage of failures in the test is going to be used to estimate the percentage of failures in actual use.

1.2 Partition Testing

Partition testing consists of selecting test cases from specific subsets of the input space. Within each subset the test cases are selected at random. The intuitive appeal of this approach is based on the assumption that the cases within a partition will be homogeneous. If one or more cases in a partition perform correctly then the other cases in the partition are expected to perform

similarly. Partition testing's effectiveness is also based on having some information about what faults are likely to exist in a program. This information allows the selection of cases that are more likely to fail. Path testing, mutation testing, data flow testing and functional testing are all examples of partition testing.

1.3 Comparison

Recent research results run contrary to most practitioners' intuition.[1] [2] Simulations and other analyses have shown that partition testing may not be much more effective than random testing at detecting faults. In some cases, partition testing may even be less effective than random testing. The research also tends to indicate that the real effectiveness of partition testing comes from focusing test cases in areas where failures are more likely to occur, rather than from the homogeneity of the partitions. Put simply, this says that you are more likely to find faults if you know where they are and use this knowledge in your testing. If you do not know where the faults are likely to be, partition testing is not significantly better than random testing. In addition, any given partitioning scheme gradually becomes ineffective. As the faults at which the scheme is targeted are fixed, fewer and fewer faults will be found.

2. Problems with Random Testing

There are two practical problems with large scale random testing. The most important is generating a large number of random inputs from the operational distribution that the software will encounter in actual use. The second is how to generate correct expected results to compare to the actual output.

2.1 Operational Distribution & Data Generation

For many programs it is easy to envision a method of generating random inputs. For example, consider the testing of a data processing type of program that updates a data base in accordance with a set of transactions. One could generate a random number of data base records containing random data in their fields. Then random transactions could be generated and run through the program. However, the purpose of testing is to find faults that will actually impact the user and to estimate the reliability of the program in actual use. To serve these purposes the data base and transactions should "resemble" actual ones. Unfortunately, it is not easy to describe what "resemble" means in this context, without knowing the true parameters of the program. Clearly there could be a "typical" proportion of invalid data, and each field could have the same

probability distribution as will occur in use. Also, any "relevant" interactions between the fields in a record could be taken into account. However, matching any of these parameters may be pointless if the program's operation is unaffected by them. If the program fails on large databases and whenever two consecutive records have the same key value then these are the parameters where the sample data must "resemble" the actual data and the distribution of the field values is irrelevant.

In other cases random generation of input data is hard to envision. Take the example of a C compiler. Randomly generating strings of characters is not difficult, but would probably only test the error reporting capabilities of the compiler. Randomly generating strings that mostly obey the syntax of the C language is more difficult but still possible. Automated tools that can accomplish this are available. Randomly generating strings that "resemble" actual C programs is, for practical purposes, impossible.

2.2 Expected Results

Another difficulty with random testing is the work involved in generating the expected results for the random inputs. In the examples above, manually determining the correct output would be very difficult and error prone. Failing to notice that an input should be handled as a special case is particularly likely and also particularly dangerous since the program may fail to test for the same special case. When the program that is being tested is replacing an existing program it may be possible to use the existing program to determine the expected results. However, it is rare that a new program is designed to produce exactly the same output as another program. Exact imitation is usually the goal only for direct ports from one hardware environment to another. A new data processing system is developed to handle some situations better than the old system and to handle additional data fields. A new compiler is expected to generate more efficient object code, support new commands, etc. Regardless of these problems, a pre-existing program may be of some assistance if it can be determined which cases, fields, etc. it can and cannot handle.

For partition tests, determining the expected results can also be time-consuming and lead to errors. However, partition tests that are focused on a particular feature tend to be smaller and simpler than actual user inputs. This makes manual preparation of expected results more practical and accurate. Also, since each case is focused on a particular feature, there is a good chance that the only difficult part of each example is the part that is emphasized as the partition criteria. This makes it harder to miss conditions that need special handling.

3. Testing of Printer Controllers

3.1 Description of the Product

The product that was being tested in this example was the Phaser™ printer controller. This controller accepts PostScript®, a postfix language that contains over 200 commands. PostScript® has many features of a general purpose procedural language and can be used as one. Therefore, testing the controller's language interpreter is similar to testing other language interpreters or compilers. The controller card on which the software resides plugs into a PC bus and emulates a serial and a parallel port. Color and monochrome print engines can be connected to the controller.

3.2 Advantages for Random Testing

Random testing of the controller was made practical by the fact that there already existed a large number of printer drivers that serve as automatic generators of PostScript® programs. In actual use, very few customers write PostScript® programs by hand. Printer drivers in graphics applications packages are the main generators of PostScript® programs and write a very different style of PostScript® from that written by human beings. Each driver tends to use only a small subset of the features of the language. Most drivers generate a "prolog" that redefines the language to provide a more compact format with a limited, specialized set of features. The data to produce the output is then sent in this new language. By primarily using programs generated by printer drivers, we were able to draw a large number of random inputs from a distribution that is similar to the actual operational distribution.

Another advantage that the printer controller software had for using random testing was the relative ease of producing expected results. Since the graphics applications programs also drive other printers and since the desired picture can be seen on the screen of a personal computer, oracles were readily available that helped determine whether the output was correct. This advantage was somewhat offset by the fact that comparing printed output cannot be easily automated and requires tedious and somewhat error prone visual comparison.

3.3 Random Test Procedure

For the random tests we gathered sample pictures from the developers of graphics software packages, from graphics artists, from customers and from within Tektronix. We did not attempt to do a scientific sampling of pictures but used nearly every example we could find. Since most

PostScript® is generated from applications packages, most of the samples were from these packages. We also used some PostScript® programs that had been manually written to illustrate certain special effects. The samples included a wide range of types of pictures. Text from word processors, line drawings from CAD packages and complex filled graphics from drawing programs were included. Shaded images from scanned input were also used and many of the pages included mixtures of the above. We captured output from about 100 combinations of applications and drivers, with a total of about 1500 test files and over 2000 pages of output. The files were sent to the controller by an automated testing system that added some header information to each page and logged each file as it printed. All error messages returned by the controller were logged, as were other detectable failures.

3.4 Partition Test Procedure

We also carried out functional testing involving about 500 test files. Each of the commands of the language was tested under several input conditions. Within a group of similar commands, such as the math operators, all error and overflow conditions were tested at least once. However, it was not practical to generate cases for all conditions on all commands at the system test level. More complete testing had been done at the unit test level for most operators. The functional test files were handled by the same test system as the random test files.

4. Comparison of Test Results

The random tests were running before the functional tests, due to problems with the design of the functional tests and differences in the emphasis placed on the two types of tests. Therefore, useful comparisons of the number of faults found by the two methods cannot be made. The types of faults found can be compared as long as the differences in timing are kept in mind.

Fault and failure data were taken from the bug tracking system that was used to handle all reported failures of the product. The bug tracking system was used for failures resulting from errors in the specification, design, user documentation and implementation tasks.

Two analyses were made on the failures reported to the bug tracking system. The first was to examine whether the failures found by the functional tests were related to the intent of the tests. The second was to compare the severity of the failures detected by the two types of tests.

The failures selected for analysis were limited to those related to the subsystem that generated text characters. This subsystem was selected because it had the most thorough functional

testing. The functional test of the text subsystem was also running earlier than the other functional tests.

4.1 Functional Tests are Actually Random

There were 32 bug reports resulting from the functional tests of the text subsystem. In terms of their relation to the purpose of the test that caused them, they fell into three groups: directly related, indirectly related and unrelated. The first group of reports described failures which were clearly related to the intent of the test case. An example from this group was a case that tested that the font data dictionaries were protected against modifications. The dictionaries were found to be writable. Since any test case that tried to write to a font data dictionary would have found the failure, this partition is "homogeneous". All of the directly related bug reports involved test cases that were selected from homogeneous partitions. This group contained nine bugs, or about 28% of those reported from the text functional tests.

The second group of reports contained failures that were only indirectly related to the purpose of the test case. A common example in this group was that of a specially modified character (obliqued, for example) that failed to print because of a fault in the graphics subsystem. While the graphics fault was exposed by the obliquing of the character, it was not the obliquing mechanism itself that failed. Other cases, many not even related to text, could have failed because of the same fault. Homogeneity in the indirectly related bugs varied widely. This group also contained nine of the reported bugs.

The third group contained failures which had no relation to the purpose of the test. An example from this group was the printing of fonts in different sizes, using a loop that incremented the font size. Due to rounding problems, the reported font sizes were incorrect. The math problem was unrelated to the font sizing, and other methods of printing the fonts in different sizes would have worked correctly. This group contained 14, or about 44% of the reported bugs.

The above data indicate that the functional tests served largely as random tests. Over 70% of the reported bugs were not in the section of code that was supposed to be tested by the test. In terms of total failures the percentage is even higher since the faults that were directly related to the purpose of the test case generally caused only one or two failures. The indirect and unrelated faults often caused many test cases to fail.

4.2 Random Testing Finds More System Crashers

Each of the bug reports in the bug tracking system was classified according to the severity of the failure. System crashes that required user intervention to restart the controller were considered to be the most severe failures. Incorrect output and unexpected error messages were the second most severe failures. Performance problems were third and possible enhancements were fourth. The table below shows the number of bug reports received for each category .

	Crash	Incorrect	Performance	Enhancement	Total
Functional Test -- Related	0	6	2	1	9
Functional Test -- Indirect	2	5	1	1	9
Functional Test -- Unrelated	2	10	2	0	14
Functional Test -- Total	4	21	5	2	32
Random Test	18	51	4	0	73
Total	22	72	9	2	105

In percentages:

	Crash	Incorrect	Performance	Enhancement	Total
Functional Test -- Related	0.0	66.7	22.2	11.1	100.0
Functional Test -- Indirect	22.2	55.6	11.1	11.1	100.0
Functional Test -- Unrelated	14.3	71.4	14.3	0.0	100.0
Functional Test -- Total	12.5	65.6	15.6	6.3	100.0
Random Test	24.7	69.9	5.5	0.0	100.0
Total	21.0	68.6	8.6	1.9	100.0

The functional tests were less likely to find the most severe bugs than were the random tests. Chi-square for the differences between total functional test bugs and random test bugs is 8.86, significant at the 95% level. If only the bugs that were directly and indirectly related to the

purpose of the test are counted as due to functional testing and the others are considered to be due to random testing the differences between the bug severities are significant at the 99% level.

These results are consistent with the observation in [2] that partition testing finds faults if the person devising the partitions has a correct idea of where the faults are likely to be. In looking for performance problems and incorrect results the functional test approach of trying special cases, boundary conditions and "things I would have forgotten" appears to work as expected. However, functional testing appears to have been less effective at detecting faults that cause crashes. In the case of the printer controller, the typical failures that crashed the controller were due to unintended memory sharing. The underlying fault was often an uninitialized pointer or writing outside of array bounds in low level routines. Failures due to these faults are clearly going to be harder to predict at a functional level.

4.4 Project Management Implications

The results of the random tests were particularly useful in the management of debugging. The faults that caused the most obvious failures or the greatest number of failures in the random testing were generally fixed first. In addition, the user base of an affected application could be considered in deciding priorities. This allowed a more focused approach to preparing for preliminary software releases since it ensured that the users receiving a particular release would not encounter known problems in using it. Faults that caused failures during partition testing were more difficult to prioritize. Regardless of the severity of the partition test failures, it was necessary to guess at how often they would occur in actual use since the partition tests provided no data about this.

It was also possible to estimate the reliability of the system in user applications and to track improvements in this reliability over time. No form of partition testing could supply this information. If the partition tests were successful in focusing test cases in partitions that were more likely to fail, the failure rate for partition tests would be higher than that expected in actual use.

Random testing was also important for validating (or belatedly making) design decisions involving quality tradeoffs. In some cases, improving the appearance of one class of output would detract from the appearance of another class. Partition tests provided no information to help decide between the options. Without application generated input, there was no way to know how often each case would occur in actual use, nor how offensive the problems would be in the cases where they did occur.

5. Conclusions

The observations concerning the differences and similarities of the two approaches need to be confirmed or contradicted by further analysis and data from other projects. The author's experience has been that the tendency of functional testing to serve largely as random testing occurs on most software projects. The differences in the severity of the detected failures may be more project dependent.

The experience of this project shows that random testing can be used for testing of software and that it has a number of advantages over partition test methods. A primary advantage of random testing is that it focusses testing in a way that will improve the usability of the product. For finding the types of severe failures found on this project, the random tests were more effective than the partition tests. Random testing allows reliability estimation and assists in project management with information that is not available from partition testing.

If certain conditions can be satisfied, random testing should be considered as a substitute for partitioned methods or as a complement to them. The key conditions include a source of input data from the expected operational distribution and a reliable means of determining the correct results.

References

1. J. Duran and S. Ntafos, An evaluation of random testing, *IEEE Transactions in Software Engineering*. SE-10 (July, 1984), 438-444.

2. D. Hamlet and R. Taylor, Partition Testing Does Not Inspire Confidence, *Proceedings, Second Workshop on Software Testing , Verification and Analysis*, Banff, 1988, 206-215.

WHAT WILL IT COST TO TEST MY SOFTWARE?

Elaine J. Weyuker

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, New York 10012

Abstract

This paper describes an empirical study to determine the cost of using a previously proposed family of data flow test data adequacy criteria. The goal of the study was to establish the practical usefulness of these criteria and to provide a means of predicting the size of test sets needed to adequately test a given program.

Elaine J. Weyuker is an Associate Professor of Computer Science at the Courant Institute of Mathematical Sciences of New York University. Her research interests include software testing and validation, software metrics, and software engineering.

Seventh Annual Pacific Northwest Software Quality Conference
Portland, Oregon
September 1989

This research was supported in part by the National Science Foundation under grants CCR-85-01614 and CCR-88-02210, and by the Office of Naval Research under Contract N00014-85-K-0414.

WHAT WILL IT COST TO TEST MY SOFTWARE?

Elaine J. Weyuker

Introduction

A family of software test data adequacy criteria was proposed in [RW82] and [RW85] and theoretical complexity analysis described in [W]. These criteria are based on the data flow characteristics of the program being tested. Data flow testing criteria require the selection of test data that exercise certain paths from a point in a program where a variable is defined, to points at which that variable definition is subsequently used. By varying the required combinations of definitions and uses, a family of test data selection and adequacy criteria was defined.

Every variable occurrence in the program is assigned a category. Besides distinguishing between *definitions* (variable occurrences at which a variable is given a new value, as in an input statement or the left-hand side of an assignment statement) and *uses* (variable occurrences at which a variable is not given a new value), we distinguish between two distinct types of uses: predicate uses (p-uses) and computation uses (c-uses). *P-uses* are uses in the predicate portion of a decision statement such as **while...do**, **if...then...else**, or **repeat...until** statements. *C-uses* are uses which are not p-uses, including variable occurrences in the right-hand side of an assignment statement, or an output statement.

The criteria require that test data be included which cause the traversal of subpaths from a variable definition to either some or all of the p-uses, c-uses, or a combination of p-uses and c-uses of that variable. The *all-definitions* criterion requires sufficient test data such that every definition is used at least once. Another way of viewing this criterion is that it requires that test data be included which cause the traversal of at least one subpath from each variable definition to *some* p-use *or some* c-use of that variable definition. The *all-c-uses* criterion requires that test data be included which cause the traversal of at least one path from each variable definition to *every* c-use of that variable definition. The *all-p-uses* criterion requires that test data be included which cause the traversal of at least one path from each variable definition to *every* p-use of that variable definition. The *all-uses* criterion requires that test data be included which cause the traversal of at least one subpath from each variable definition to *every* p-use *and every* c-use of that variable definition. The *all-du-paths* criterion requires that test data be included which cause the traversal of *every simple* subpath from each variable definition to *every* p-use *and every* c-use of that variable definition. The relationships between these criteria are explored in [RW82] and [RW85]. Precise definitions of the entire family of criteria are included in [RW85] for a very simple language, and in [FW88] for Pascal.

Once the family of criteria had been defined, we proved that, at least from a theoretical point of view, the criteria have desirable characteristics. We then investigated the (theoretical) complexity of each of the criteria. We demonstrated in [W] that, *in the worst case*, all but one of the criteria require at most a number of test cases which is quadratic in the number of decision statements of the module being tested. We further

demonstrated that these upper bounds were "tight" by constructing programs which actually require the specified worst case number of test cases. When we considered these "worst case programs", however, we recognized that they were not "natural" programs and we hypothesized that "real" programs would require far fewer test cases to satisfy the data flow criteria. In fact our intuition said that even though in theory a program might require as many as 2^d test cases for a program containing d decision statements in order to satisfy the all-du-paths criterion, in practice the relationship was likely to be linear.

This led us to begin hand simulations to apply our theory to test many programs in order to get a feel for its usefulness in detecting errors and evaluating the comprehensiveness of proposed test sets. Since this was done manually, most programs in these informal experiments were of necessity small, but we found the results encouraging enough to build a tool, ASSET, to automate the application of this theory. In addition, since we view these ideas as primarily applicable to unit testing when individual modules are generally of modest size, even the small programs which we manually tested were not wholly unrealistic.

The initial version of ASSET [FWW], [FW85] accepted programs written in the very limited subset of Pascal of the original theory [RW82], [RW85]. We have now extended the theory to full Pascal [FW88], and currently ASSET accepts programs written in (almost) full Pascal. The theory has also been extended to be used for regression testing [OW], and ASSET is currently being modified to provide a data flow regression testing facility.

Once ASSET was available for programs written in Pascal, rather than the original "toy" language, we used ASSET to evaluate the test sets for many sample programs. This experience strengthened our conviction that using data flow information as the basis of a test data adequacy criterion is useful and sound. At this point we felt that systematic empirical studies to evaluate the data flow testing criteria were essential.

In this paper we present the results of an empirical study which was designed to investigate the costs of using various data flow testing techniques in practice. There are two main goals of this study. The first is to confirm our intuition and informal experience that our family of software testing criteria based on the program's data flow characteristics is practically usable. That is, we gathered evidence that even as the program size increases, the amount of testing, expressed in terms of the number of test cases sufficient to satisfy a given criterion, remains modest. We explore, in this study, several ways of evaluating this hypothesis.

The second goal is closely related to the first: to provide the prospective user of these criteria with a way of predicting the number of test cases that will be needed to satisfy a given criterion for a given program. Again we consider several plausible bases for such a prediction.

The Design of the Experiment

One difficulty in designing software engineering experiments in general, and testing experiments in particular, is choosing appropriate software to be studied. We selected the

suite of programs, "Software Tools in Pascal" [KP] by Kernighan and Plauger, which consists of over 100 routines. We tested most of the modules in the suite, using most of the criteria of our family, in order to establish empirical complexity estimates.

Testers were instructed to select test cases using the selection strategy of their choice. They were not encouraged to select test cases explicitly to satisfy the data flow criteria. Their goal was to do a comprehensive job of testing and to apply the data flow criteria to evaluate the adequacy of the test set once they believed they were done testing. We also decided in this study that no attempt would be made to minimize the number of test cases used in each test set. Instead, "atomic" test cases were selected. That is, rather than attempting to cleverly select a test case which might fulfill several characteristics simultaneously, testers were instructed to select "natural" test cases, each with a single purpose. Thus, for example, rather than selecting a test case which begins with at least two blank characters, is at least ten characters long, containing at least one repeated non-blank character, given that these three characteristics are determined to be significant, we would instead select three distinct test cases, each displaying only one of the characteristics. This was done because testing practitioners frequently do not attempt to minimize test sets. In addition, when regression testing is performed, having each test case have a single, clearly identifiable role facilitates this type of testing. Also, the use of "atomic" test cases yields pessimistic results. We felt this was important since if non-minimal test sets which satisfy the data flow criteria were of manageable size, then the criteria are certainly practically usable.

Although ASSET was applied to almost all of the subprograms in the Kernighan and Plauger suite, we considered in this study only those subprograms with five or more decision statements. There were 29 such programs used in this study which are listed in the Appendix. For programs with fewer decision statements, the upper bounds on the number of test cases needed to satisfy the criteria are so small that there is no difficulty running that number of test cases. Only as the program size grows (as assessed by the number of decision statements) does it become important to determine whether substantially fewer test cases suffice in practice to satisfy the criteria.

If the selected criterion requires the traversal of unexecutable subpaths, then no test set, regardless of how extensive, will satisfy the criterion and thereby adequately test the program. In such a case, we shall say that a test set *almost satisfies* a data flow criterion provided it causes all the *executable* subpaths required by the given criterion to be traversed. In [FW86, FW88], we introduced a new family of adequacy criteria, known as the feasible data flow criteria, which are based on the data flow criteria but require only that the *executable* definition/use associations be exercised. Since it is clearly impossible to cause unexecutable paths to be executed, we consider a criterion satisfied when a test set almost satisfies the criterion. In the remainder of this paper, we shall say that a criterion has been satisfied provided that it has been almost satisfied.

Results

The primary questions addressed in this study involve the cost of using the data flow testing criteria, as assessed in a variety of ways. In [W] we demonstrated the following

theoretical upper bounds on the number of test cases required to satisfy each of the data flow criteria. If d is the number of (two-way) decision statements in the program, then all-c-uses, all-p-uses, and all-uses require at most $(d^2 + 4d + 3)/4$ test cases, and all-du-paths requires at most 2^d test cases. Furthermore, for each of these bounds other than all-c-uses, we constructed a program for which the only test sets which can satisfy the criterion are at least the size of the stated upper bound. Of course, test sets may exceed the size of the upper bound and still not satisfy a given criterion if the test cases have not been judiciously selected or if there are unexecutable definition/use associations.

In this study we consider the "empirical complexity" for the criteria all-c-uses, all-p-uses, all-uses, and all-du-paths. We did not consider the cost for the all-definitions criterion, except in comparison to the more expensive criteria, since for this criterion the upper bound on the number of test cases is quite moderate. Since the theoretical upper bounds were expressed in terms of the number of decision statements in the program, we have used that as our basis for the empirical study as well.

Given the previously stated goals of this study, we tested 29 programs from the Kernighan and Plauger suite and applied each of the considered data flow criteria. We then computed:

1) the least squares line: $t=\alpha d+\beta$, where t is the number of test cases sufficient to satisfy the given criterion for the subject program, and d is the number of decision statements in the subject program

2) the weighted average of the ratios of the number of decision statements in a subject program to the number of test cases sufficient to satisfy the selected criterion

3) the maximum value of the ratio of the number of test cases sufficient to satisfy the selected criterion for a given subject program to the number of decision statements in that program

4) the weighted average of the ratios of the theoretical upper bound on the number of test cases needed to satisfy the selected criterion for a given subject program to the number of test cases sufficient to satisfy the selected criterion

5) the weighted average of the ratios of the number of test cases sufficient to satisfy the all-definitions criterion to the number of test cases sufficient to satisfy the all-uses criterion

6) the weighted average of the ratios of the number of test cases sufficient to satisfy the all-definitions criterion to the number of test cases sufficient to satisfy the all-du-paths criterion

7) the weighted average of the ratios of the number of test cases sufficient to satisfy the all-uses criterion to the number of test cases sufficient to satisfy the all-du-paths criterion.

The results of 1, 2, 3, and 4 are summarized in TABLE 1. The results for 5, 6, and 7 are presented in TABLE 2.

	all-c-uses	all-p-uses	all-uses	all-du-paths
least squares	t=.52d+1.87	t=.76d+1.01	t=.81d+1.42	t=.93d+1.40
weighted average	t=.43d	t=.70d	t=.72d	t=.81d
max t/d	3.5	2.33	3.67	3.67
avg percent up bd	13%	23%	24%	0.4%

TABLE 1

We see that for each of the criteria considered, when we express t (the number of test cases) as a function of d (the number of decision statements in the subject program), either by computing the least squares line or weighted average, the coefficient is less than 1. These results are consistent with our intuition that the relationship between the number of test cases and the program size was truly a linear one. These results are shown in lines 1 and 2 of TABLE 1, representing the two different ways of computing this relationship. The fact that the tester can expect to need a very modest number of test cases in order to do quite a comprehensive job of testing is especially surprising and encouraging for the all-du-paths criterion which requires in the worst case a number of test cases which is exponential in the number of decision statement in the program.

As mentioned above, by looking at these results, testers have a rough guide to the number of test cases they can expect to need on average in order to satisfy a given criterion. This permits testers to select the most comprehensive criterion their budget permits.

It is also important to consider the maximum value of $\frac{t_i}{d_i}$ encountered for each criterion in the study. Even though the average case value is moderate, it is important to determine whether there were *any* cases encountered which required exceptionally large numbers of test cases. Practically, this ratio represents an empirical worst case for each of the data flow criteria considered.

For the 29 programs of the Kernighan and Plauger suite, the maximum value of $\frac{t_i}{d_i}$ for the criterion all-c-uses, was 3.5. For the all-p-uses criterion, the maximum value of $\frac{t_i}{d_i}$ was 2.33, for the all-uses criterion the maximum value of $\frac{t_i}{d_i}$ was 3.67, and for the all-du-paths criterion the maximum value was 3.67. Note that in each case, this ratio was small, giving us further evidence that the data flow criteria are usable in a practical setting. These values represent an "empirical worst case" for each criterion, and are a

sharp contrast to the theoretical worst case results described in [W] and mentioned above. They are shown on line 3 of TABLE 1.

It is interesting to note that in this study, the same program was responsible for the empirical worst cases for each criterion. That is, for each of the ways of assessing adequacy which we considered, the same program required the greatest amount of testing relative to its size.

Another way to consider the data is to compare the actual number of test cases sufficient to satisfy criteria to the theoretical upper bounds. They show that in practice only a moderate number of test cases were needed for each of the criteria, even when the theoretical upper bound indicated that a huge number of test cases might be needed. The weighted averages of these values for each of the criteria is shown in line 4 of TABLE 1.

The final question we considered was: "How much more expensive is it, in practice, to use the more demanding criteria, rather than the weaker criteria?" The answer, surprisingly, was that, on the average, it was not much more difficult to satisfy the most demanding criteria as compared to the least demanding criterion. To determine this, we considered, for each subprogram of the study, the ratio of the number of test cases sufficient to satisfy the all-definitions criterion to the number of test cases sufficient to satisfy all-uses, as well as the ratio of the number of test cases sufficient to satisfy the all-definitions criterion to the number of test cases sufficient to satisfy the all-du-paths criterion. Finally, we considered the ratio of the number of test cases sufficient to satisfy the all-uses criterion to the number sufficient to satisfy the all-du-paths criterion. Recall that the all-uses criterion has a theoretical upper bound which is quadratic in the number of decision statements in the program, whereas the all-du-paths criterion has a theoretical upper bound which is exponential in the number of decision statements. These results are presented in TABLE 2. In each case we see that if the cost of testing is assessed in terms of the number of test cases sufficient to satisfy the criterion, it is hardly more expensive to satisfy a very demanding and comprehensive criterion than a much weaker one. This was quite a surprising and encouraging result.

average all-uses/all-defs	1.79
average all-du-paths/all-defs	1.96
average all-du-paths/all-uses	1.08

TABLE 2

We have recently received additional data provided by Shimeall [SL] [S]. He applied the all-p-uses criterion to a suite of eight numerical programs as part of a study to compare the effectiveness of various testing techniques. The specification for these programs was derived from a specification provided by TRW.

Since Shimeall only considered the all-p-uses criterion, we were only able to compute the items of TABLE 1 for these programs. The results for these programs were even more encouraging than those for the Kernighan and Plauger suite. This was particularly surprising since Shimeall's programs were generally larger and more complex and were written to satisfy a more sophisticated specification than those of the Kernighan and Plauger suite. We had expected that as program size and specification size increased, the programs would become harder to test. This was not the case, and at present we have no explanation for this phenomenon.

The results for these programs are presented in TABLE 3. We see that the actual number of test cases needed was, in general, substantially below the values predicted by our least squares analysis and weighted average. In addition, the actual numbers of test cases sufficient to satisfy the all-p-uses criterion never exceeded 1% of the number predicted by the theoretical worst case analysis. Unlike the Kernighan and Plauger data, the number of test cases never exceeded the number of decision statements for these programs. In the worst case encountered, the ratio of the number of decision statements to the number of test cases was only .60, as compared to the ratio of 2.33 for the Kernighan and Plauger programs using the all-p-uses criterion.

number decision stmts	number test cases	pred lst sqs	t/lst sqs	pred wt avg	t/wt avg	upper bound	t/up bd	t/d
434	107	331	.32	304	.35	47524	.002	.25
209	103	160	.64	146	.71	11138	.009	.49
196	115	150	.77	137	.84	9801	.012	.59
325	110	248	.44	228	.48	26732	.004	.34
246	103	188	.55	172	.60	15376	.007	.42
301	113	230	.49	211	.54	22952	.005	.38
334	117	254	.46	234	.50	28224	.004	.35
173	104	133	.78	121	.86	7656	.014	.60

TABLE 3

Conclusions and Future Work

We have found the results of this empirical study to be very encouraging and confirming of our intuition that the data flow criteria are cost effective to use. In particular, we found that although the theoretical upper bounds on the number of test cases needed to satisfy most of the data flow criteria are quadratic or exponential, in practice only small numbers of test cases (as compared to the program's size) were

needed to satisfy the criteria. Generally, if a program contains d decision statements, a test set of size less than d was sufficient to satisfy any of the data flow criteria.

Additionally, our data provides the tester with a way of predicting how many test cases should be needed to satisfy a given criterion for a given program. It was also encouraging to see that using the most demanding criterion, all-du-paths, required only, on average, a factor of less than two times the number of test cases as the very undemanding criterion, all-definitions. All of these results provide evidence that the data flow criteria are usable in practice.

One limitation of using the Kernighan and Plauger suite as the basis for this empirical study is that a large percentage of the programs in the suite are similar in type: namely string processing programs. For this reason, we were very interested in the results for the all-p-uses criterion for the Shimeall programs since they were relatively large numerical programs. We would like to extend these experiments to include many more programs, with a greater variety of types of software. It would be interesting to study the data flow criteria relative to database programs and determine whether there are substantially different results than obtained in this study and also to consider additional numerical programs and all of the data flow criteria for such programs.

A limitation of the Kernighan and Plauger suite of a different nature is that they are well designed and highly modularized. As a result, almost all of the subprograms are relatively small in size. These are not, however, toy programs. We do not know, however, if substantially larger, poorly designed programs would generally require larger amounts of test data relative to their size than the Kernighan and Plauger programs required. Based on the preliminary results using the Shimeall programs, however, it does not appear that this is the case.

We intend to extend this research in a number of different directions. One of the most important ones is to extend the theory of data flow testing to other languages, notably C and Ada. An important issue for C is the pervasive use of pointers. Currently, ASSET treats pointers in Pascal as purely syntactic objects. While this is generally adequate in Pascal, substantial thought will have to go into considering the most appropriate way to deal with them in C. Several other issues must also be considered for a C theory, including the limited block structure supported by C, variable declarations within a compound statement, and the treatment of compound conditionals. Ada will present a greater number of problems. Two important issues that must be considered are how does the theory change in the face of concurrency, and how should exception handling be dealt with? Other important language features to consider include packages and parameter passing rules.

Once the theory has been extend to other languages, we expect to modify the implementation of ASSET to accomodate these languages. Currently ASSET parses the Pascal subject program. We plan to explore the usefulness of changing this so that ASSET will first translate a subject program into an intermediate language, and then operate on this intermediate code. In that way, once the theory has been extended to C and Ada (and perhaps other Algol-like languages), only a front-end parser would be needed in order to provide ASSET to testers of such programs. Of course it would then

be necessary for a single intermediate language to be appropriate for all of these languages.

Another facility we expect to be able to provide involves allowing the tester to enter not only inputs as test cases, but (input,expected output) pairs. ASSET would then be able to act as an oracle and report to the user when an actual output differs from the expected output. In such an instance, ASSET could provide the tester with the path traversed. ASSET will then be usable as a debugging aid, as well as a test data adequacy assessment tool.

Acknowledgments

This study would not have been possible without the members of the Software Testing Group at Courant and the support of the National Science Foundation and the Office of Naval Research. In particular, ASSET was implemented largely by Phyllis Frankl, Stewart Weiss, and Ernie Campbell. The test cases were designed and run by Valerie Barr, Ernie Campbell, Biing-Chiang Jeng, and Peter Fryscak. Tom Ostrand made many helpful suggestions both about the analysis of the data and the presentation of the results. Tim Shimeall kindly provided the all-p-uses data for his program suite. I am indebted to all of these people.

APPENDIX

Programs Considered in the Study

AMATCH
APPEND
ARCHIVE
CHANGE
CKGLOB
CMP
COMMAND
COMPARE
COMPRESS
DODASH
EDIT
ENTAB
EXPAND
GETCMD
GETDEF
GETFN
GETFNS
GETLIST
GETNUM
GETONE
GTEXT
MAKEPAT
OMATCH
OPTPAT
RQUICK
SPREAD
SUBST
TRANSLIT
UNROTATE

REFERENCES

[FW85] P.G. Frankl and E.J. Weyuker, "A Data Flow Testing Tool," *Proceedings of IEEE Softfair II*, San Francisco, Dec. 1985.

[FW86] P.G. Frankl and E.J. Weyuker "Data Flow Testing in the Presence of Unexecutable Paths," *Proceedings of the Workshop on Software Testing*, Banff, July 1986, pp. 4-13.

[FW88] P.G. Frankl and E.J. Weyuker "An Applicable Family of Data Flow Testing Criteria," *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, October 1988, pp. 1483-1498.

[FWW] P.G. Frankl, S.N. Weiss and E.J. Weyuker, "ASSET: A System to Select and Evaluate Tests", *Proceedings of the IEEE Conference on Software Tools*, New York, April 1985.

[KP] B.W. Kernighan and P.J. Plauger, *Software Tools in Pascal*, Addison Wesley, Reading Ma., 1981.

[OW] T.J. Ostrand and E.J. Weyuker, "Using Data Flow Analysis for Regression Testing," *Proceedings Sixth Annual Pacific Northwest Software Quality Conference*, Portland, Or, Sept. 1988, pp. 233-248.

[RW82] S. Rapps and E.J. Weyuker, "Data Flow Analysis Techniques for Program Test Data Selection," *Proceedings Sixth International Conference on Software Engineering*, Tokyo, Japan, Sept. 1982, pp. 272-278.

[RW85] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Trans. Software Eng.*, Vol. SE-11, No.4, April 1985, pp. 367-375.

[S] T.J. Shimeall, "An Experiment in Software Fault Elimination and Fault Tolerance", Ph.D Thesis, U. of California, Irvine, to appear.

[SL] T.J. Shimeall and N.G. Leveson, "An Empirical Comparison of Software Fault Tolerance and Fault Elimination," *Proceedings Second Workshop on Software Testing, Verification, and Analysis*, Banff, July 1988, pp. 180-187.

[W] E.J. Weyuker, "The Complexity of Data Flow Criteria for Test Data Selection," *Information Processing Letters*, Vol. 19, No. 2, August 1984, pp. 103-109.

Testing Motif

Jason Su
Hewlett-Packard
Corvallis Information Systems
1000 NE Circle Boulevard
Corvallis, Oregon 97330

Printed: July 28, 1989

"*I want us to achieve a tenfold improvement in...software quality over the next five years.*"
John Young, Vice President of Hewlett-Packard, April 24, 1986.

In January, 1989, the Open Software Foundation announced their User Environment Component, Motif. Motif would be a combination of technologies bringing together Hewlett Packard's 3D visuals with a Presentation Manager type look and feel. This would be based on Hewlett Packard's window Manager and DEC's toolkit and widget library. Hewlett Packard, Corvallis Information Systems was contracted to develop Motif for OSF. CIS agreed to apply Hewlett Packard's Software Quality practices. "Testing Motif" describes the tools, process and release criteria.

Jason Su received a Bachelor's degree in Computer Science from the University of California at San Diego in 1982. He has worked at Hewlett Packard, Corvallis Division since March, 1983 as a Research and Development software engineer. Currently Jason is the Integration and Test team project leader for HP's future widget library and windowing manager.

1. Introduction

The schedule for Motif could be categorized as 'fast track'. Investigation to Release spannned a period of only seven months. Bi-weekly snapshots containing current development were sent to OSF from week two. In the end, project Motif delivered the following items to OSF:

- Motif Xt and Xm - DEC's instrinsic and widget library was retrofitted with Hewlett Packard's 3D visuals. Widgets would also received a Presentation Manager 'look and feel'. And most importantly, a formal suite was created to insure quality on what originally had only been "monkey" and "beta" tested.

- Motif Mwm - Hewlett Packard's window manager was modified to take advantage of Motif's new widget library. It was also retrofitted with Presentation Manager supplemented for additional features.

- Motif Style Guide - Hewlett-Packard also provided a style guide document that specified the basic look and feel of Motif from the user's perspective. It describes PM-compatible behavior and 3D appearance and their use. Application programmers use this guide to insure that users be exposed to a uniform interface.

Three teams were created to address the requirements: Xm/Xt development, Mwm development, and Test/Integration. While the development teams' responsibilities are outside the realm of this paper, the Test and Integration team released snapshots, built nightly and snapshot libraries and window manager, generated tests, architected a formal test suite, and tracked defects.

Acknowledgements go to the rest of the test team, Jim Espeland, Paul Ritter, John Bertani, Sharada Bose, and manager, Frank Hall. Their commitment to the Motif testing effort has gone far beyond the call of duty.

2. Tools

2.1 Xpeek - Test client builder

Previous toolkit testing at Hewlett-Packard, Corvallis Information Systems (CIS), uncovered a need for an easy way to generate "C" source clients. Whereas the standard archaic method was to generate one or more mammoth programs to "test" a product, it was found that smaller, more malleable test programs that concentrated on a particular functionality were easier to create and maintain. To speed the generation of these test clients, Xpeek was developed. Through the use of on-screen menus and forms, a user would interactively build Motif widget interfaces.

In developing Motif, a toolkit rich in widgets, the natural progression was to utilize it in our test tools if possible. Thus Xpeek itself became a Motif client which includes and has the ability to exercise every widget and gadget. The Xpeek interface to the widgets coded to interrogate individual widget's for their resources and values. This served a two-fold purpose. Xpeek did not have to undergo source modifications when widget specifications changed (e.g. resource names or default values), New widgets were easily incorporated. Only a recompilation was necessary. Secondly, when a test was generated, all resources were presented for changing. It was not necessary to refer to the External Specification for widget resources.

One of Xpeek's major features was its ability to generate "C" source code that corresponded to the interactive creation of the desired widget interface. Standard functions such as creating a top level shell or widget, and setting resource values would generate "C" source and declaration statements that were pumped into a .c and .h file. The following is a sample of standard Xpeek output that generates a Paned window with four arrow buttons.

2.1.1 *sample.h*

```
/****************************************
*     Automatic test program generation     *
*                Version 1.4                 *
*          Mon Jul 17 12:49:57 1989          *
****************************************/


/* Standard C headers */
#include <stdio.h>
#include <sys/signal.h>

/* Xm meta class header */
#include <Xm/XmP.h>

/* Xt header */
#include <X11/Shell.h>

/* Xm headers */
#include <Xm/ArrowB.h>
#include <Xm/ArrowBG.h>
#include <Xm/BulletinB.h>
#include <Xm/CascadeB.h>
#include <Xm/CascadeBG.h>
#include <Xm/Command.h>
#include <Xm/CutPaste.h>
#include <Xm/DialogS.h>
#include <Xm/DrawingA.h>
#include <Xm/DrawnB.h>
#include <Xm/FileSB.h>
#include <Xm/Form.h>
#include <Xm/Frame.h>
#include <Xm/Label.h>
#include <Xm/LabelG.h>
#include <Xm/List.h>
#include <Xm/MainW.h>
#include <Xm/MenuShell.h>
#include <Xm/MessageB.h>
#include <Xm/PanedW.h>
#include <Xm/PushB.h>
#include <Xm/PushBG.h>
#include <Xm/RowColumn.h>
#include <Xm/Scale.h>
#include <Xm/ScrollBar.h>
#include <Xm/ScrolledW.h>
#include <Xm/SelectioB.h>
#include <Xm/SeparatoG.h>
```

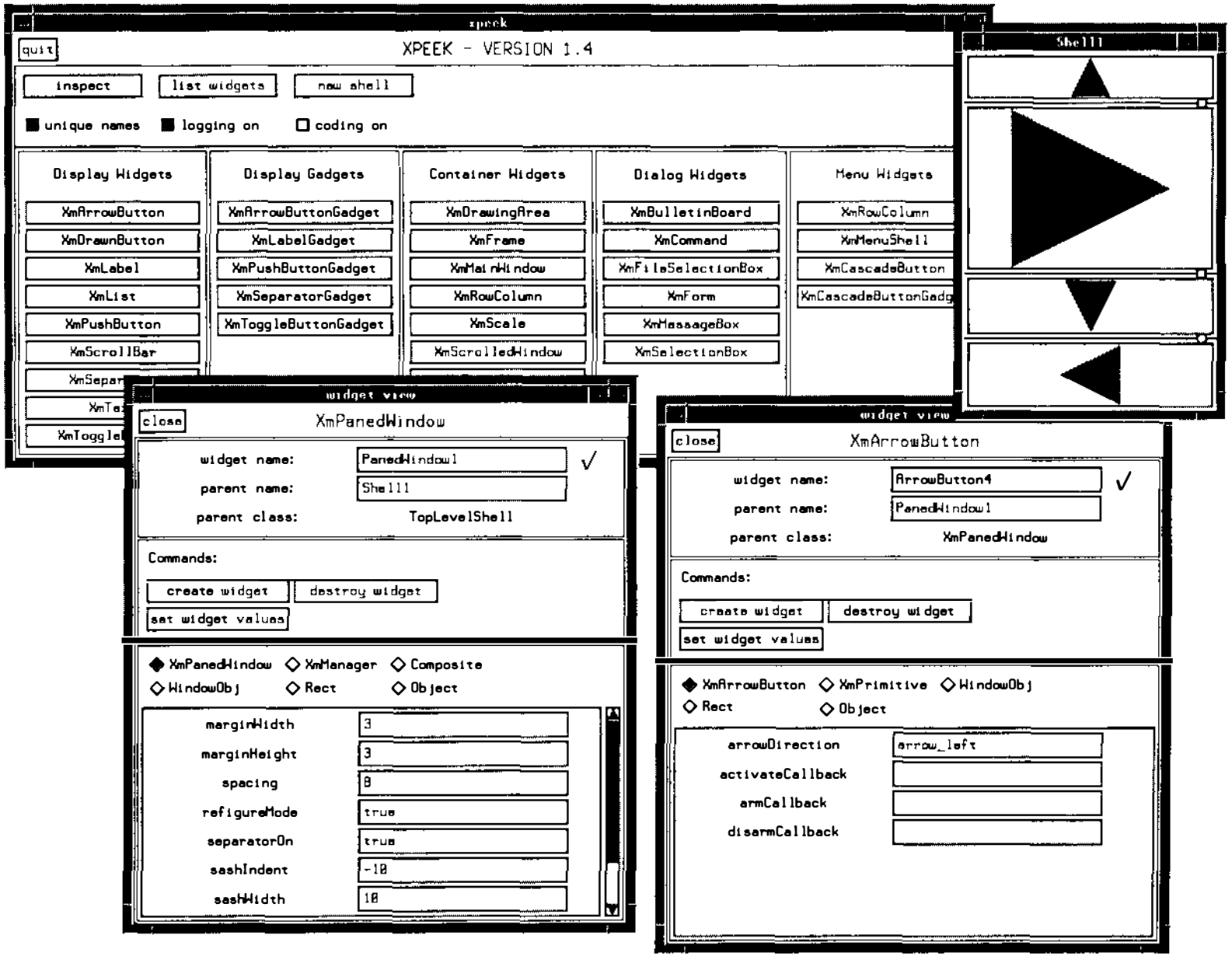
```
#include <Xm/Separator.h>
#include <Xm/Text.h>
#include <Xm/ToggleB.h>
#include <Xm/ToggleBG.h>

#define TRACE(string)  if (trace) {printf(string);}

#define PROGNAME  "sample"
#define MAX_ARGS  100

/*  Global Variables  */
Display  *display;
Widget    Shell1;
Widget    PanedWindow1;
Widget    ArrowButton1;
Widget    ArrowButton2;
Widget    ArrowButton3;
Widget    ArrowButton4;

/*  Private Functions  */
static void          GetOptions();
static void          Quit();
```

2.1.2 *sample.c*

```
/****************************************
*    Automatic test program generation     *
*                 Version 1.4              *
*          Mon Jul 10 10:49:57 1989        *
****************************************/


#include "sample.h"

void  main(argc, argv)
   int     argc;
   char  **argv;
{
   Boolean       trace = False;
   register int  n;
   Arg           args[MAX_ARGS];
   XmString      tcs;

   signal(SIGHUP,  Quit);
   signal(SIGINT,  Quit);
   signal(SIGQUIT, Quit);

   XtToolkitInitialize();
   display = XtOpenDisplay(NULL, NULL, argv[0], "XMclient",
                     NULL, 0, &argc, argv);
   if (display = = NULL) {
      fprintf(stderr, "%s:  Can't open display0, argv[0]);
```

```
    exit(1);
  }

  GetOptions(argc, argv, &trace);

  n = 0;
  XtSetArg(args[n], XmNwidth,  400);  n++;
  XtSetArg(args[n], XmNheight, 300);  n++;
  XtSetArg(args[n], XmNallowShellResize, True);  n++;
  Shell1 = XtAppCreateShell(argv[0], NULL, applicationShellWidgetClass,
                    display, args, n);
    TRACE("XtSetArg: XmNwidth 4000);
    TRACE("XtSetArg: XmNheight 3000);
    TRACE("XtSetArg: XmNallowShellResize0);
    TRACE("XtAppCreateShell: Shell1 ApplicationShell0);

  XtRealizeWidget(Shell1);
    TRACE("XtRealizeWidget: Shell10);

  n = 0;
  PanedWindow1 = XmCreatePanedWindow(Shell1, "PanedWindow1", args, n);
  XtManageChild(PanedWindow1);
    TRACE("XmCreatePanedWindow: PanedWindow1  parent: Shell10);

  n = 0;
  ArrowButton1 = XmCreateArrowButton(PanedWindow1, "ArrowButton1", args, n);
  XtManageChild(ArrowButton1);
    TRACE("XmCreateArrowButton: ArrowButton1  parent: PanedWindow10);

  n = 0;
  XtSetArg(args[n], XmNarrowDirection, XmARROW_RIGHT);  n++;
  ArrowButton2 = XmCreateArrowButton(PanedWindow1, "ArrowButton2", args, n);
  XtManageChild(ArrowButton2);
    TRACE("XtSetArg: XmNarrowDirection arrow_right0);
    TRACE("XmCreateArrowButton: ArrowButton2  parent: PanedWindow10);

  n = 0;
  XtSetArg(args[n], XmNarrowDirection, XmARROW_DOWN);  n++;
  ArrowButton3 = XmCreateArrowButton(PanedWindow1, "ArrowButton3", args, n);
  XtManageChild(ArrowButton3);
    TRACE("XtSetArg: XmNarrowDirection arrow_down0);
    TRACE("XmCreateArrowButton: ArrowButton3  parent: PanedWindow10);

  n = 0;
  XtSetArg(args[n], XmNarrowDirection, XmARROW_LEFT);  n++;
  ArrowButton4 = XmCreateArrowButton(PanedWindow1, "ArrowButton4", args, n);
  XtManageChild(ArrowButton4);
    TRACE("XtSetArg: XmNarrowDirection arrow_left0);
    TRACE("XmCreateArrowButton: ArrowButton4  parent: PanedWindow10);

  TRACE("XtMainLoop0);
  XtMainLoop();
}
```

```
static void  GetOptions(argc, argv, trace)
   int          argc;
   char        **argv;
   Boolean      *trace;
{
   register int i;

   for (i = 1; i < argc; i+ +) {
      if (argv[i][0] = = '-' && argv[i][1] = = 'v')
         *trace = True;
      else {
         fprintf(stderr, "0sage:  %s [-v]0, PROGNAME);
         fprintf(stderr, "        The -v option enables trace output.0);
         exit(1);
      }
   }
}


static void  Quit()
{
    printf("Begin exiting ... please standby ... 0);
    fflush(stdout);
#ifdef  BFA
    ExitBFA();
#else
    exit(0);
#endif  /*  BFA  */
}
```


xpeek
quit
XPEEK - VERSION 1.4
inspect
list widgets
new shell
unique names
logging on
coding on
Display Widgets
XmArrowButton
XmDrawnButton
XmLabel
XmList
XmPushButton
XmScrollBar
Display Gadgets
XmArrowButtonGadget
XmLabelGadget
XmPushButtonGadget
XmSeparatorGadget
XmToggleButtonGadget
Container Widgets
XmDrawingArea
XmFrame
XmMainWindow
XmRowColumn
XmScale
XmScrolledWindow
Dialog Widgets
XmBulletinBoard
XmCommand
XmFileSelectionBox
XmForm
XmMessageBox
XmSelectionBox
Menu Widgets
XmRowColumn
XmMenuShell
XmCascadeButton
Shell1
widget view
close
XmPanedWindow
widget name:
PanedWindow1
parent name:
Shell1
parent class:
TopLevelShell
Commands:
create widget
destroy widget
set widget values
XmPanedWindow
XmManager
Composite
WindowObj
Rect
Object
marginWidth
3
marginHeight
3
spacing
8
refigureMode
true
separatorOn
true
sashIndent
-10
sashWidth
10
widget view
close
XmArrowButton
widget name:
ArrowButton4
parent name:
PanedWindow1
parent class:
XmPanedWindow
Commands:
create widget
destroy widget
set widget values
XmArrowButton
XmPrimitive
WindowObj
Rect
Object
arrowDirection
arrow_left
activateCallback
armCallback
disarmCallback

2.2 BFA - Branch Flow Analysis

Hewlett Packard has developed a tool based upon the article "Software Verification Using Branch Analysis" co-authored by Daniel E. Herington, Paul A. Nichols, and Roger D. Lipp. The Branch Flow Analysis tool simply instruments source code to enable automatic data collection. The data collected is the run time count of branches taken as a program executes.

For Motif, a preprocessor pass is executed that inserts a "bucket", or counter, after each branch into the desired source code. A path to the BFA database file is compiled into the code as well. Thereafter, the code is compiled normally to generate an instrumented version. This was done on a weekly basis for Motif's Xt and Xm widget libraries and the Mwm window manager.

This is a sample code fragment of a simple routine before and after bfa preprocessing:

```
foo()
{
   if(1) {
      i=1;
   } else {
      i=0;
   }
}
```

***

```
foo()
{
   if(! _bfa_init)
      _InitializeBFA();

   _bfa_array[0]++;
   {
      if(1) {
         _bfa_array[1]++;
         {
            i=1;
         }
      } else {
         _bfa_array[2]++;
         {
            i=0;
         }
      } _bfa_array[3]++;
   }
}
```

As the test program runs, control flow passes through the instrumented code, incrementing branch count buckets. Exiting the program causes a support routine to be called to update the database file. Another report program interrogates the database file and prints program, file, and branch statistics.

2.2.1 *BFA branch report segment*

BFA Version CND B.00.00 <Alpha Database Search>

BFA Database Statistics Summary Report Mon Jul 10 10:22:43 1989
Database file name - bfadbase

	Total #	# Hit	% Hit
Files	50	50	100.00%
Procedures	1476	1463	99.12%
Branches	14132	12754	90.25%

BFA Database Branch Report Mon Jul 10 10:22:43 1989
Database file name - bfadbase
Source file name - Command.c
Procedure name - Initialize

Branch #	Type	Line #	# Hits	Not Hit
1	entry	321	17	
2	if	327	1	
3	fallthru/327	330	17	
4	if	333	1	
5	fallthru/333	336	17	
6	if	338	1	
7	while	340	2	
8	fallthru/340	343	1	
9	fallthru/338	347	17	
10	if	350	1	
11	fallthru/350	351	17	

100.00% of Branches Hit

BFA indicates what areas of code have or have not been tested. It provides managment with the ncessary statistics to track progress. It's primary purpose for the test team was to uncover areas of code that had not been tested. However, BFA does not insure all possible paths have been covered, only that at least one has been executed. BFA also does not insure correctness! The tests must take sufficient data points and a tester or some mechanism must still evaluate the results.

Organizations should resist the temptation to use BFA as a goal instead of an aid. Testing 85% of the easiest code could still result in the majority of defects surfacing in the hardest 15%. In addition, should BFA be too highly emphasized, gathering branch metrics may become a higher priority than actually testing the logic.

Branch flow analysis goals are usually stated in a triplet, e.g. 85/100/100. The first figure refers to the percentage of branches hit at least once. The second and third figures state the percentage of internal routines and external entry points hit. These last two figures are commonly 100%, indicating that all internal and external functions should be tested in some way.

2.3 McCabe

During the investigation phase of project Motif, the HP-CIS Quality and Productivity group made available a McCabe tool from "McCabe and Associates". The McCabe tool generates a "Cyclomatic Complexity Metric" which reflects the number of basis paths through a section of code. A basis path is characterized by the fact that:

- Each statement is hit at least once.
- Basis paths can be ordered such that every possible independent path can be built.

The claim for McCabe's cyclomatic complexity is that it seems to more accurately indicate a program's complexity over simply basing it on size or number of independent paths. For instance, large programs may seem complex but may actually consist of straight line code that is easy to follow. Conversely, a small program with highly nested if-statements may generate a very high number of independent paths. Basing complexity on total number of independent paths may reach infinity for programs containing loops.

In practice, the cyclomatic complexity may be applied to software projects in the following ways:

- The number may be used to police the code complexity. Industry experience has shown that limiting code to a cyclomatic complexity of 10 has significantly reduced the number of defects received throughout the software lifecycle. The one exception is the case statement which inherently reports a high complexity because of the large number of arbitrary paths.
- The cyclomatic number has been shown to correlate to defect density. Thus, it can be used to predict the sections of code with a high potential for defects. Additional attention in development and concentrated testing can be aimed at these sections.

The tool offered by McCabe and Associates provided additional features considered by project Motif to be useful. Although standard Unix commands exist to generate call-to listings, this tool generates call-by listings as well in a more comprehensible format. Taking this one step further, the tool also generates a graphical representation of a call-by/call-to chart called a "Battlemap". From a specified entry point (e.g. main()), a graph is created showing the procedural calls and their depths. This was a tremendous help in decreasing the learning curve of new code.

2.4 X Test Monitor

Xtm, X Test Monitor, is a record-and-playback mechanism. It captures the tester's keyboard and mouse actions and their time intervals and places them in a file. During a regression test, it will play back the events in exactly the same sequence. Screen and window dump/matches may be requested to ensure that the results occuring from the input actions are correct. A typical scenerio used by the Motif project was to: execute a test client, exercise some functionality (like pushing a toggle button), and take a screen dump/match.

Xtm depends on an "input synthesis" X server extension to bypass the normal keyboard and mouse devices at the device dependent (ddx) layer of the server. During the recording phase, it gathers server data including input events and their time intervals, and window and screen dumps. During playback, the events are reinserted into the server input queue at the same time intervals and the window and screen dumps are matched for correctness.

Capturing screen dumps can become expensive in terms of disc space. Xtm recognized this problem and optionally will checksum the image instead of capturing the raw image. Unfortunately when a checksum match fails, one cannot match the captured image against the match image. Operator

intervention is required to rerun the test to determine the mismatch reason. Even so, with motif, there were almost 200 xtm scripts containing an average of 10 screen dumps for a display with 1280x1024x8 resolution. The disc requirements for full screen matches would have been 2 gigabytes.

Another feature of xtm is its ability to convert from an xtm script to a human read/writeable format and back again. Thus, minor changes can be made to correct keystroke errors, change time intervals, etc.

Xtm was developed at HP-CIS and has been contributed to the public domain. Xtm should be available through the X Consortium or your computer vendor.

2.5 CMETRIC - C Style Analysis and HP Metric program

Cmetric is a program modeled after a paper by Michael J. Rees, "Automatic assessment aids for Pascal programs". Although HP's cmetric program reports content and style information, its use for the Motif project was to count Non-Commented Source Statements. NCSS is a factor in calculating Defect Density, one of our test release criteria.

2.6 DTS - Defect Tracking System

Although DTS, Hewlett Packard's Defect Tracking System, refers to the database and tools that store, categorize, notify, and retrieve defects, it is only part of the task of handling defects. Like so many defect databases that are currently in use, it requires the submitter to insert the following information:

- Severity - acceptable values are from 0 (user misunderstanding), 1 (cosmetic defect. usable but could be better), 3 (usable with a simple workaround or fix), 5 (usable with moderate workaround), 7 (much harder to use than reasonably expected, 9 (can't use the product; no workaround exists).

- Description - a brief one line description

- Symptom - one of system crash, deadlock/hang, unexpected abort, incorrect behavior, interface to software, data lost, file disappearance, documentation wrong, infinite loop, inconsistent behavior, message unclear or wrong, interface to O.S., unfriendly, enhancement request, documentation missing, other

- OS release id - version

- Related file - allow the submitter to fully describe his problem, attach source files, etc.

The responsible engineer who resolves the defect adds the following information:

- Resolution type - code change, specification change, documentation changed to disallow action, documentation changed to support action, duplicate report, forward, no change, design change, operator error, change deferred, duplicate fix, not reproducible, user defined, other

- Phase Introduced - investigation/specification, design, integration and test, implementation, post release

- Phase Found - same options as above

- Phase Fixed - same options as above

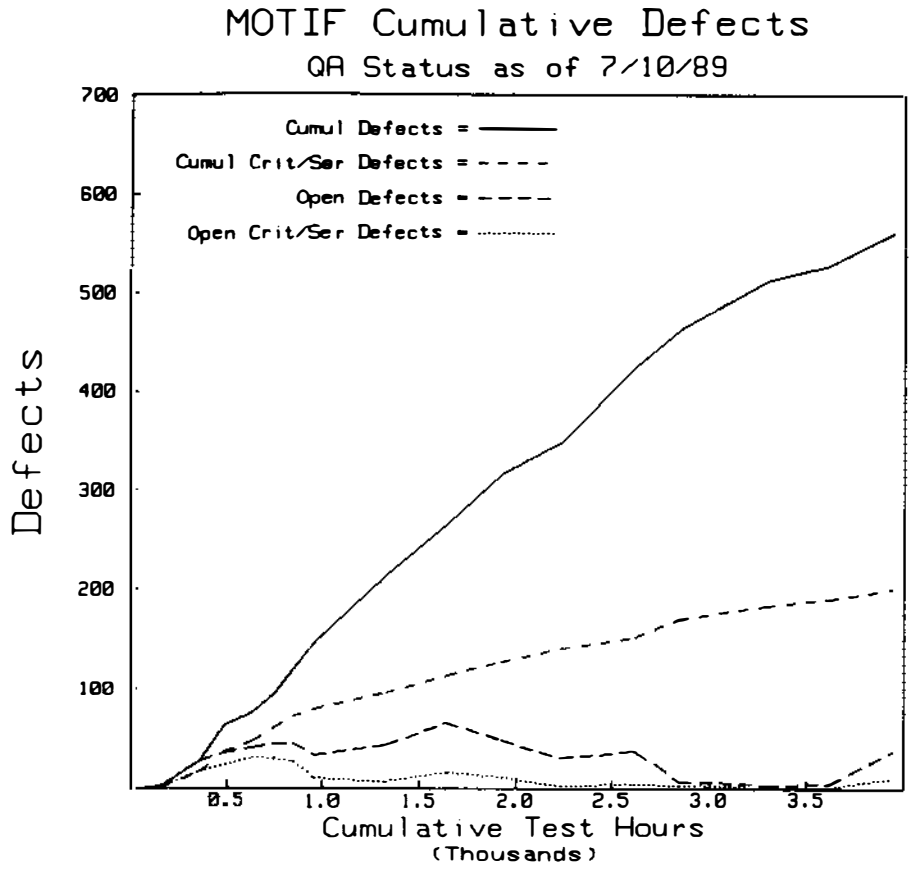
Various report modes exist to notify management and engineers. Management summaries contain a list of defects by responsible engineer, status (resolved or open), and severity. Motif managers were automatically emailed a report each morning via cron. Engineers are notified by email when a defect is submitted and they are assigned as the responsibile engineer. Daily, Motif engineers were sent a list of open defects that fell within their responsibility. Engineers then could get detailed reports. In addition, detailed reports are submitted to an HP internal notes group for general perusing by the masses.

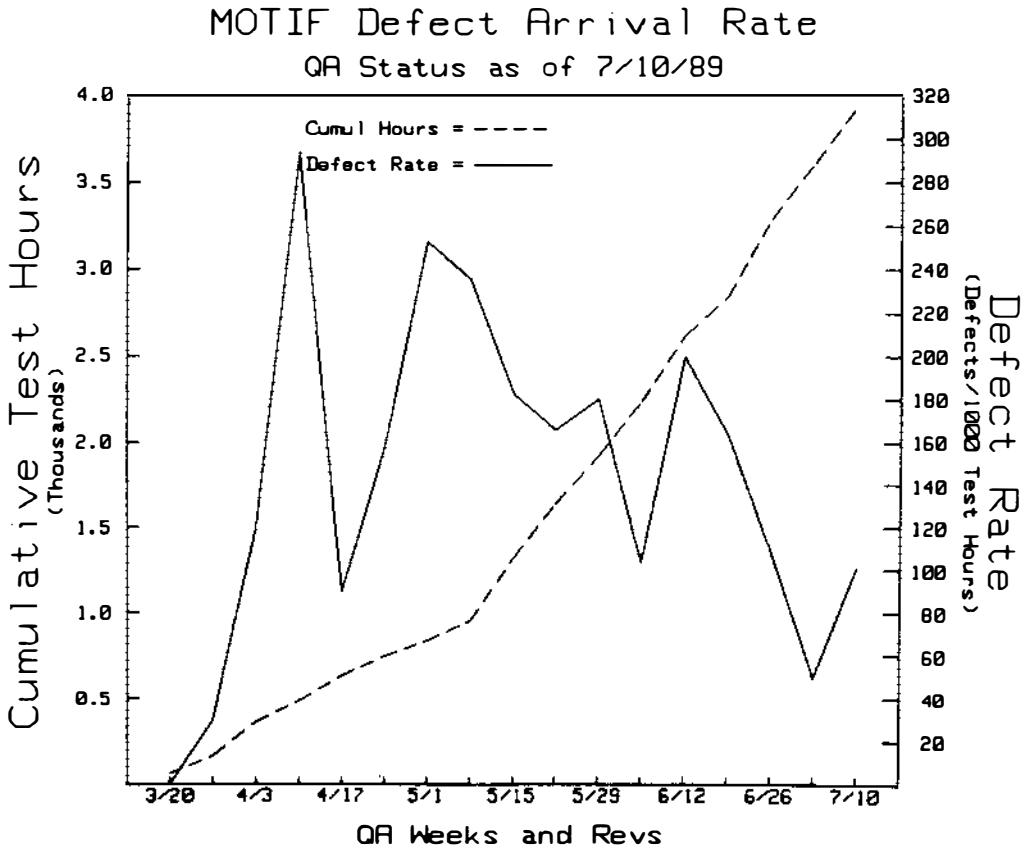
On a weekly basis, a "Software Quality Status" table, "Cumulative Defect" chart and "Defect Arrival Rate" chart are distributed to the Motif team. The "Software Quality Status" table contains the current BFA, breadth, defect density, and KNCSS (thousand lines of non-commented source). The charts give the Cumulative Defects and Arrival Rate as a plotted graph. The Cumulative Defects chart ideally should level off to zero slope at the end of the project. The Defect Arrival Rate should taper off to the project goals by the end of the project. Refer to the body of this paper for the Motif quality goals.

2.6.1 *Software Quality Status Table*

Motif
SOFTWARE QUALITY STATUS
(Level 'C' Certification Goals)

Component	BFA DEPTH	BREADTH		DEFECT DENSITY			CODE SIZE
	(Branches)	Internal (Proced.)	External (Targets)	Critical	Serious	Med+Low	KNCSS / %
	(Goal=85%)	(Goal=100%)		(Goal=0)	(Goal=0)	(Goal=1)	of Motif
Xm	87%	100%	100%	0.0	0.0	0.3	63.4 / 61.4
Xt	84%	100%	100%	0.0	0.0	1.2	16.3 / 15.8
Mwm	87%	100%	100%	0.0	0.4	0.0	23.5 / 22.8


MOTIF Cumulative Defects
QA Status as of 7/10/89
Cumul Defects =
Cumul Crit/Ser Defects =
Open Defects =
Open Crit/Ser Defects =
Defects
700
600
500
400
300
200
100
0.5
1.0
1.5
2.0
2.5
3.0
3.5
Cumulative Test Hours
(Thousands)


MOTIF Defect Arrival Rate
QA Status as of 7/10/89
Cumul Hours =
Defect Rate =
Cumulative Test Hours
(Thousands)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
320
300
280
260
240
220
200
180
160
140
120
100
80
60
40
20
Defect Rate
(Defects/1000 Test Hours)
3/20
4/3
4/17
5/1
5/15
5/29
6/12
6/26
7/10
QA Weeks and Revs

3. Process

3.1 Test Plan

The test plan served as the initial step for the test group. It was designed to be a guide in which the testing effort was conceived and specified as a whole. The intent was to describe the how-to's and what's to direct the test team to its goals.

The plan formally stated the testing goals.

- 85/100/100 BFA (Branch Flow Analysis)
- Zero (known) Critical defects by CRT (Change Review Team phase)
- 1 or less medium or low severity open defect per 10 KNCSS (thousand non-commented source statements) by CRT
- Defect Arrival Rate less than 50 per 1000 test hours by CRT
- Automated Formal Test Suite

Motif test goals, in general are born from Corporate guidelines. The limits are based on John Young's mandate of a ten fold improvement of software quality by 1991. The metrics, then, are approximately a half way point to that of three years ago.

As described previously, the intent was to hit 85% of the branches in the code. The Test team would make sure that all internal and external targets were hit as well.

All known critical defects would be resolved for Change Review Team. Change Review Team is a phase where representatives from each development and test group must thereafter approve changes in the code. Elements of risk (side affect), severity, and impact to other components (test suite, clients) are considered for approval.

The Defect Density and Arrival Rate are gauges for the malfunctioning of a product and the readiness for its use. A defect density of 1 medium to low severity open defect per 10 KNCSS has been determined to be the current threshold for Hewlett-Packard. Defect Arrival Rate is plotted against 1000 test hours logged. The rate should decline toward the end of the development cycle and remain stable to indicate that 95-99% of the defects have been found. A threshold of 50 new defects found/1000 test hours logged has been determined to be the current threshold for Hewlett-Packard.

The test plan also has the responsibility of mapping out the tools and the process in which they would be used. Again, it's not so important to predict every minute detail, but rather to hit on the high points and lay down a global groundwork.

The test plan should specify an agenda detailing tasks that are to be accomplished by certain dates. This usually follows alpha, beta, and release time dates.

Motif's test plan also included the initial architecture for the the formal test suite which included a description of the test environment. The .mwmrc and .Xdefaults configuration files were part of this description.

Dependencies should be clearly specified. For Motif, disc space, machine resources, staffing resources, and the requirement for xtm and the accompanying server input synthesis extension were dependencies.

Features to test should be as well outlined as possible. For Motif, this was leveraged heavily off of the External Specification.

3.2 Evaluation Phase

During the evalution phase, the test team awaited the first Motif libraries which would contain minimal functionality to begin test generation. The time was spent in testplan creation, building and acquiring new tools, and evaluting current test technology for possible reuse.

At this time, because of Motif's fast track schedule, the test team realized that the time invested in generating tools would increase productivity in the crucial stages when time would become a precious commodity. Xpeek started out as an interactive client builder that was written around HP's Xw widget library. Through the use of icons, menus, and the mouse, it provided a graphical interface through which the user could easily generate widget interfaces and source code. In its early age, it was written to rigid specifications with hard-coded widget classes and resources. This implied that each time a widget was added or changed, Xpeek would undergo intensive rework and testing. By porting to Motif, an opportunity arose to improve Xpeek.

The major contribution to the successfulness of the new Xpeek was its use of the Xt Intrinsic's XtGetResourceList(). Through this call, all widget resource values *and their names* could be obtained. Xpeek was crafted such that when the user requested the "widget view" of a widget, the resource names and values were displayed in a general manner. Adding a new widget (that the development team finished or built to minimal functionality) was a simple matter of adding the new widget class name to a table and recompiling. Changes in resource names, or the addition/subtraction of resources was a simple Xpeek recompile.

Xpeek was also fitted with features to Get and Set resource values. Through on-screen forms, resource values could easily be set. Through it's inspect feature, selecting an existing widget peeked into it through the use of XtGetResources() and popped up a resource form. Reverse converters were used to change the 1's and 0's into human readable names and values.

There have been a few projects within Hewlett-Packard that attest to the success of McCabe's Cyclomatic Complexity number. In these cases it has been shown that when strictly adhering to keeping code complexity below 10, the level of defects in the test and post release phases is lower than non-policed projects. Because of Motif's fast track schedule, forcing engineers to redesign and rewrite modules was unreasonable. Instead, the test team provided the complexity numbers to the development team which helped pinpoint modules for formal code reviews.

The test team also helped the development teams to use the tool to generate call-by/call-to listings and graphical charts. This enabled the engineers to reduce the learning time for the existing code that they were to modify.

An investigation was launched to determine the feasability of reusing existing test suites for hpwm, HP's window manager and Xw, HP's widget toolkit. Because the Motif window manager was a port of hpwm, about one third of the tests came straight across. New features including PM behavior, ICCCM conformance, icon box, and menu accelerators required new tests to be generated. HP's widget toolkit suite was too different to leverage tests. The differences in names for routines and resources, and the discrepancies in functionality were sufficient reason for regenerating the suite.

3.3 Test Development Phase

After the first snapshot, the toolkit team generated a document indicating the percentage functionality of each widget. The test team began generating tests for those widgets that were complete or very nearly complete. The toolkit team updated this "functionality" document weekly up until alpha for which 100% functionality was required.

At this time, widget testing responsibility was divided amongst the test team. Using Xpeek, clients were created to exercise all resources. Initially, they were linked to BFA instrumented libraries and exercised by hand. Actions included iconifying/uniconifying, selection, destroying, etc. A BFA report was then generated and the test client source was supplemented by hand to reach additional branches or a new client was created. The goal was to reach a minimum of 85% branches for each widget or module. In addition, 100% of routines were entered at least once.

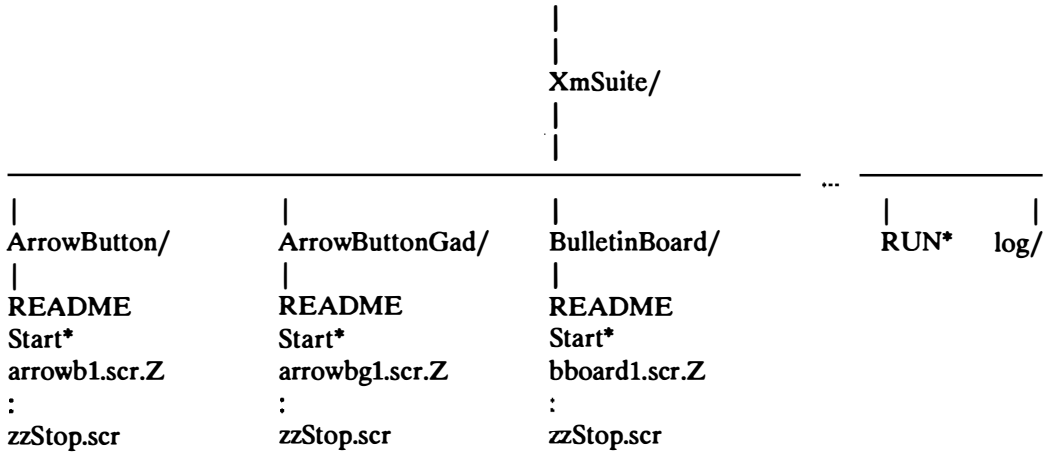
After the test client was created and the tester learned how to manipulate the client to reach the maximum branch coverage, the test was incorporated into the formal suite. As changes to the libraries occurred, regression testing could be performed automatically using the tests in the formal suite. The goal was that it would no longer be necessary to have an operator rerun old tests for each release. Another advantage was that the process would be formalized and future testing would guarantee the same level of testing.

The test suite would take advantage of the hierarchial filesystem of Unix. Directory trees were created for each widget with Bourne shell scripts controlling the execution.

Actual test clients reside in a common directory to allow access by any of the test scripts. The desire is to run the test suite in a standard environment to insure consistent results. This consistency had to apply to the display environment as well as the run time environment. The initial display/test environment would have Mwm and one xterm running only. Each directory contains a Start script and configuration files for this reason. These files happen to be commonly linked to one another to save file space. Xtm scripts (*.scr[.Z]) and potentially other support files make up the rest of the directory contents.

"Start" is a Bourne shell script that is the fuse for each individual test directory. Its responsibility is to actually set the test environment and spawn off the individual xtm test scripts. Start prepends the 'clients' directory to the current path. The current widget test directory becomes the HOME directory so local configuration files are used. It then starts mwm, sets the background to lightgrey, and initiates a term window. Start then cycles through the xtm scripts, first uncompressing them and then passing them to the xtm tool. Each of the xtm scripts clears the term window and starts the desired test client by typing "clear" and the test name into the term window. Thereafter the xtm script exercises the client, taking screen captures at strategic moments to check for correctness. Before exiting the "Start" script the environment is restored to the pre-testing state by executing the xtm script zzStop.scr. This kills the term window and exits Mwm.

The Xm test tree segment



In each widget directory, a README file describes the actions performed in the xtm script. The primary purpose of the README was to enable users who did not have xtm to manually emulate the test process. It also served as an invaluable aid in maintaining the test suite.

At the next level up, the RUN script was designed to bring together and automate the test suite. It cd's (change directory) to each of the widget directories and executes the Start script. The output from stdout and stderr are redirected to a file in the log directory with a unique start-time stamped name, eg. RLOG.0721.1405 (Run log, July 21 @ 2:05 pm).

Initially, the RUN script did not restart the server after each directory. This would cause a reliability problem that surfaced in xtm. Screen matches began failing due to inconsistent allocation of the colormap. The solution was to restart the server after each directory. The RUN script, now, cd's to the desired widget test directory and executes xinit passing it Start. This has reduced the number of inconsistent state errors.

3.4 **Regression Test Phase**

The hope was that as tests were developed, they would be rolled into the suite and would require only occasional modifications when changes to the library were made. Unfortunately, the library required frequent modifications and a large number of tests failed for each new bi-weekly snapshot. The problem stemmed from the fact that user interface software deals with an immense amount of potential display interaction. Mwm places window decorations around client windows with 'handles' and 'icons' to stretch, move, and pop up menus. Should a client window change size, the handles and icon locations change. Since Xtm reproduced mouse movements exactly as recorded, it would lose the handle or icon and the test would get lost from that point on. The only alternative is to rerecord the test.

Because the act of evaluating and rerecording a test was so expensive, it would have been best if the test team had not generated any xtm scripts until beta phase. Instead, tests could be rerun manually from the description READMEs. This would necessitate detailed READMEs (which we already had) that described expected results as well as procedure. It would have made better use of our time to commit tests to xtm after Beta phase, when all functionality was implemented and only serious or critical defects remained.

The Mwm suite, on the other hand, experienced relatively few changes. Even though a large part of the suite was leveraged from HP's window manager, Mwm had fewer changes in feature set due to the fact that it had already existed as hpwm for some time. New tests complemented enhancements that were

introduced during the normal course of development.

4. Conclusions

The ability of the test team to successfully reach its goals and on schedule can be attributed to many factors. The members were of the highest caliber. Although the simultaneous product development and test creation often broke clients and xtm scripts, the test team displayed an exemplary attitude when faced with rerecording xtm scripts. The philosophy of the test team was that it was a "service group to the development teams. It was not the test team's responsibility to restrict the development of the Motif product, but to assist."

The workstations provided to us were the fastest that HP produces. Disc space became a problem in the latter stages. Our test suite would grow to 300 Mb and had to be duplicated for each test machine configuration.

Xpeek, BFA, and McCabe were dependable and increased engineering productivity significantly. Even though a great deal of pressure was placed on the test team to immediately begin generating tests, in retrospect it was wisely spent improving and learning tools to increase later productivity. In addition, a few clients required some sort of modification as resource names changed or disappeared. A recommendation for the future is to craft tests, carefully documenting detailed procedure and results. As functionality changes diminish (HP's alpha stage), then commit to Xtm scripts.

Xtm falls short of the ultimate record and playback mechanism. Xtm presented itself as a formidable opponent to be wrestled with at times. As functionality changed causing size or behavior changes to test clients, Xtm would "lose" itself from that point on. Rerecording or modifying the Xtm script would then be necessary. A better Xtm would be object based, referencing such things as Buttons and ScrollBars as objects the way a human does.

Motif fell into the BFA metric trap at times. Reaching BFA goals sometimes took priority over the tests themselves. In management's urgency to increase branches-executed, tests were generated solely for the sake of hitting branches. BFA should not be allowed to become a management stick to justify the ends. It serves as a wonderful tool to measure testing progress but should not be used to create meaningless tests. Direct impact experienced by Motif included consuming scarce disc space and time with clients that were at least half a megabyte as well as a maintenance sink.

5. References

Hewlett Packard Company, *Software Quality and Productivity Guide*, 1987.

Hewlett Packard Company, *Spectrum Program Metrics and Certification Handbook*, 1987

Daniel E. Herington, Paul A. Nichols, and Roger D. Lipp, *Software Verification Using Branch Analysis*.

Michael J. Rees, *Automatic assessment aids for Pascal programs*, SIGPLAN Notices 17(10), October 1982.

William T. Ward, *Software Defect Prevention Using McCabe's Complexity Metric*, Hewlett Packard Journal, April 1989.

1989 PROCEEDINGS ORDER FORM

Pacific Northwest Software Quality Conference

To order a copy of the 1989 Proceedings, please send a check in the amount of $30.00 to:

PNSQC
c/o Lawrence & Craig, Inc.
P.O. Box 40244
Portland, OR 97240

After January 1, 1990, please send orders to:

PNSQC
P.O. Box 970
Beaverton, OR 97075

Name _____

Title _____

Affiliation _____

Mailing Address _____

City, State _____ Zip _____

Daytime Telephone _____