NINTH ANNUAL PACIFIC NORTHWEST

SOFTWARE QUALITY CONFERENCE

October 7 - 8, 1991

Oregon Convention Center

Portland, Oregon

TABLE OF CONTENTS

Technical Track A

Technical Track B

PREFACE

Elicia M. Harrell

Welcome to the Ninth Annual Pacific Northwest Software Quality Conference. It is rewarding to know as we approach our first decade of successful Software Quality Conferences, that we have been able to provide a forum and environment for software professionals to meet, share information, learn new ideas and acquire new skills.

As software grows more complex and becomes the key component of systems, the ability to produce reliable software is more in demand. To achieve high quality and reliability, practitioners must rely on concise and highly evolved development and evaluation processes to complete their jobs. Our 1991 keynote speaker, Alfred M. Pietrasanta, addresses his insights to the evolution of the software process, drawn from many years of experience establishing and directing software processes for IBM.

We are pleased to publish these Proceedings which contain the papers presented during the technical program. The papers represent the current thinking and practice from the United States, Canada and the United Kingdom. Nineteen papers were selected from the abstracts received from our Call for Papers. An additional four speakers representing experts in the industry have been invited to share their experience and expertise during our technical sessions.

I would like to thank Hilly Alexander and Dick Hamlet, the Program Committee Co-Chairs, for the hard work it takes to pull the program together. Their work is the basis for this Conference. Many thanks also to the members of the Program Committee for their time, energy and intelligence in refereeing the abstracts and papers. The Program Committee is the foundation of the process for putting together the Conference.

I would like to thank the remaining members of the full committees, whose names are listed in the next section, who contributed their ideas, effort and attendance to many meetings to make this Conference a success.

Finally, I want to express special thanks to Terri Moore at Pacific Agenda for being able to handle the organization and administrative tasks and at the same time, keep the committees on track and moving forward.

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Elicia Harrell - President/Chair
Intel Corporation

Steve Shellans - Vice President; Keynote
Tektronix, Inc.

Lowell Billings - Treasurer; Birds of a Feather
Infotec Development

Debra Lee - Secretary; Workshops
Intel Corporation

Dick Hamlet - Program Co-Chair
Portland State University

Hilly Alexander - Program Co-Chair
ADP

Paul Blattner - Exhibits
Quality Software Engineering

G.W. Hicks - Publicity
TSSI, Inc.

Brian Johnston - Software Excellence Award
Software Process Consultants

CONFERENCE PLANNING COMMITTEE

Sue Barlett
Mentor Graphics Corp.

Sandhi Bhide
Mentor Graphics Corp.

Kit Bradley
Canyon Crest Technology

Ann Bynum
Intel Corp.

Margie Davis
ADP Dealer Services

Dave Dickmann
Hewlett Packard

Cynthia Gens

Micheal Green

Gary Hanson
Kentrox Industries, Inc.

Warren Harrison
Portland State University

Connie Ishida
Mentor Graphics

Mark Johnson
Mentor Graphics

Bill Junk
University of Idaho

Ray Lischner
Mentor Graphics

Peter Martin
Apple Computer

Howard Mercier
Intersolv

Shannon Nelson
Intel Corp.

Dave Patterson
Mentor Graphics

Misty Pesek
Tektronix, Inc.

Ian Savage
CFI

Eric Schnellman
CDP

Bill Sundermeier
Cadre Technologies

Karen Ward

Donald H. White

Nancy Winston
Mentor Graphics

Barbara Zimmer
Hewlett Packard

PRESENTERS

Marc H. Baber
Intel Corporation
15201 N.W. Greenbrier Pkwy. C01-01
Beaverton, OR 97006
(503) 629-7697

Stephen A. Bender
The Quality Connection
4 Old Vly Road
Schenectady, NY 12309
(518) 452-8166

James M. Bleman
Computer Science Department
Colorado State University
Fort Collins, CO 80523
(303) 491-7096

Brian Casey
6 Corporate Place
PYA1E-229
Piscataway, NJ 08854
(906) 699-8988

John C. Cherniavsky
CISE/OCDA Rm 304
NSF
1800 G Street NW
Washington, DC 20550
(202) 357-7349

Dr. James S. Collofello
Computer Science Department
Arizona State University
Tempe, AZ 85287-5406
(602) 951-0596

Michael Dyer
IBM Federal Sector Division
6600 Rockledge Drive
Bethesda, MD 20817
(301) 493-1490

Noelle Evans
Mentor Graphics Corporation
8005 S.W. Boeckman Road
Wilsonville, OR 97070-7777
(503) 685-7000

Geoff J. Flamank
Dynapro Systems Incorporated
800 Carleton Court
New Westminister
British Columbia, Canada V3M 6L3
(604) 521-3962

Guillermo A. Francia III
P.O. Box 2335 CS
Department of Computer Science
New Mexico Tech
Socorro, NM 87801
(505) 835-5209

Jean Hartmann
Science Laboratories
University of Durham
Durham
DH1 3LE, United Kingdom
44-91-374-3658

Robin Jeffries 1U17
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
(415) 857-8784

William S. Junk
Computer Science Department
University of Idaho
Moscow, ID 83843
(208) 885-7530

Edward W. Krauser
S/W Engineering Research Center
Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907-2004
(317) 494-7807

Brian Marick
Motorola
1101 E. University
Urbana, IL 61801
(217) 244-0431

Aditya P. Mathur
Department of Computer Science
Purdue University
West Lafayette, IN 47907
(317) 463-7822

Joe Maybee
Tektronix
Mail Stop 63-356
P.O. Box 1000
Wilsonville, OR 97070-1000
(503) 685-3572

Andrew Oram
38 High Haith Road
Arlington, MA 02174
(617) 641-1261

Mike Perdue
Sun Technology Enterprises, Inc.
2550 Garcia Ave.
Mountain View, CA 94043
(415) 336-7115

Dr. Jeffrey Voas
Information Services Division
NASA Langley
Mail Stop 478
Hampton, VA 23665
(804) 864-8136

Caroline Wardle
National Science Foundation
1800 G Street NW
Room 304
Washington, D.C. 20550
(202) 357-7349

Nancy Winston
Mentor Graphics Corporation
8005 S.W. Boeckman Road
Wilsonville, OR 97070-7777
(503) 685-7000

EXHIBITORS

Dan Zimmerman
Advanced Software Automation
2880 Lakeside Drive Rm. 226
Santa Clara, CA 95054
(408) 492-1668

Bill Sundermeier
Cadre Technologies
P.O. Box 1309
Beaverton, OR 97075
(503) 690-1318

John Cranston
File-Protek, Inc.
P.O. Box 481
Portland, OR 97207
(503) 641-9495

Hanford Choy
Infomentor Software
984 Thatcher Drive
Los Altos, CA 94024
(415) 967-5636

Barbara Kay
Interactive Development Environment
595 Market Street, 10th Floor
San Francisco, CA 94105
(415) 543-0900

Becky Johnson
KnowledgeWare, Inc.
3340 Peachtree Road
Atlanta, GA 30326
(404) 231-8575

Tali Aben
Mercury Interactive Corp.
3333 Octavius Drive Suite 104
Santa Clara, CA 95054
(408) 982-0100

Jim Steinbacher
Oregon Software
7352 SW Durham Road
Portland, OR 97215
(503) 624-6883

Shawn Wall
Powell's Technical Bookstore
33 NW Park
Portland, OR 97209
(503) 228-3906

Teresa Harrison
SET Laboratories, Inc.
26976 S. Hwy 213
Mulino, OR 97042
(503) 829-7123

Edward Miller
Software Research, Inc.
625 Third Street
San Francisco, CA 94107-1997
(415) 957-1441

David Ouellette
Terra Pacific Writing Corp.
P.O. Box 1244
Corvallis, OR 97339
(503) 754-6043

Alfred M. Pietrasanta
Independent Consultant
10 Sparkling Ridge
New Paltz, NY 12561

Al Pietrasanta spent 32 years with the IBM Corporation, participating in, and managing, many of the major activities in the evolution of the software development process. In his keynote address, Mr. Pietrasanta will cover some of the highlights of this process evolution, from his unique vantage point:

- The genesis of a software process, working on OS/360 with Fred Brooks (author of "The Mythical Man Month").
- Teaching Software Project Management to 1000 managers in the 1960's.
- Working with Mike Fagan, originator of the Formal Inspection Process.
- Establishing standardized quality and productivity metrics in all development laboratories.
- Measuring quality and productivity across the product line.
- Setting up an international Software Quality Assurance organization.
- Selling a strategy for an integrated set of process tools.
- Presenting to corporate executives an annual "State-Of-The-Process".
- Directing an institute to teach 7000 professionals modern software engineering methodology.

For these and other vignettes, Mr. Pietrasanta will draw on his experience in such IBM positions as Director of the Programming Process, Director of Systems Assurance, and Director of the IBM Software Engineering Institute.

Since retiring from IBM in 1987, Mr. Pietrasanta has consulted with several major software vendors and government agencies, and has lectured in the United States, Europe, South America and Japan on every aspect on software engineering management and software process improvement. He is presently a part-time Member of the Technical staff of the Carnegie Mellon Software Engineering Institute.

Comparing the Effectiveness of Software Development Paradigms: Spiral-Prototyping vs Specifying

William Junk
Computer Science Department
University of Idaho
Moscow, ID 83843
208-885-7530
E-mail: costest @ idui1.csrv.uidaho.edu

Mr. Junk is an Assistant Professor of Computer Science specializing in software engineering and has a particular interest in teaching and research in software quality assurance, software design, and software management areas. He has been responsible for initiating the senior-level software engineering practicum, and developing courses in Software Engineering, Software Quality Assurance, Software Metrics, and Software Process Management. Several of his courses have been offered through the facilities of the National Technological University. During 1989-1990 he was a sabbatical professor in Software Quality Engineering for Varian Medical Equipment where he focused on software development process issues. Prior to joining the University of Idaho he served in many engineering, software development, and management positions with the Space Division of General Electric Company.

Paul Oman
Computer Science Department
University of Idaho
Moscow, ID 83843
208-885-7219
E-mail: oman @ ted.cs.uidaho.edu

Mr. Oman is an Associate Professor in the Computer Science Department. He was instrumental in establishing the department's Software Engineering Lab and provides overall direction of the lab's operation. His teaching and research interests lie in the software engineering area with special emphasis on software metrics and software paradigm evaluation. He has also been extensively involved in teaching the group project portion of the software engineering practicum. He has been a member of the *IEEE Software* Editorial Board and editor of the Software Test Lab column.

Grant Spencer
Varian Medical Equipment
Software Quality Engineering
3045 Hanover Street
Palo Alto, CA 94304-1129
415-424-4516

Mr. Spencer is a Software Quality Engineer for Varian Medical Equipment. He recently completed his MSCS degree at the University of Idaho where he was involved in the evaluation of projects performed in the senior-level software engineering practicum. At Varian he is now responsible for qualification testing and V&V activities associated with linear accelerators used in cancer therapy.

Keywords: Software development paradigms, software quality, prototyping, waterfall model, spiral model, software project management, quantitative studies, programming teams.

Comparing the Effectiveness of Software Development Paradigms: Spiral-Prototyping vs. Specifying

William Junk and Paul Oman
University of Idaho

and

Grant Spencer
Varian Medical Equipment

ABSTRACT

For years there has been debate over which software development paradigm is best. There are many anecdotal reports extolling the advantages of prototyping over specifying approaches, but few controlled studies have been performed to quantify the differences between them. In this paper we describe a series of controlled experiments comparing spiral-prototyping to specifying in academic software development projects. We found that the prototyped products were completed with less effort, had lower complexity metric values, had fewer reported defects, and were rated higher on the customer's subjective evaluation of quality. We also found that management of the spiral-prototyping process is a critical element in project success or failure. Because of the experimental controls employed in our study and the realism of the programming projects performed, we believe that these results are valid equally outside the academic environment.

SOFTWARE DEVELOPMENT PARADIGMS

An issue that faces the manager of every software development project is what overall development strategy to use. In the management arena probably no other single issue has generated as much discussion. Many different strategies, software development life cycle models, and development paradigms have been proposed. Each approach has its advocates and each is accompanied by an attendant set of advantages and disadvantages. In applying these approaches, varying degrees of success have been reported.

At the center of the debate is the software development process model. The principle use of process models has been to prescribe a sequence of actions that need to be carried out during development. The purpose of a process model should be to help make software development a more reliable, predictable, and productive process.

Early process model representations were drawn from perceived parallels in hardware or system development and as a result represented software development as a sequential set of independent steps. Their representations were simplistic and lacked flexibility. It seems that no single model fits all situations and it is important to recognize the circumstances that may favor a particular approach. In his analysis of software engineering methodologies, Barry Boehm [Boeh84] identified three dominant paradigms for software development: code-and-fix, specifying, and prototyping. In the following sections we will briefly review the important characteristics of these approaches.

The Code-and-fix Paradigm

Code-and-fix software development consists of writing some code and then fixing the problems which are sure to arise, then repeating the process again. The system is built with minimal or no

specifications and with superficial design. The resulting product is initially constructed and then reworked until users are satisfied or a decision is made to cancel the project. With no emphasis on documentation there is little likelihood that any useable record of the requirements or design will be produced. Although there are probably developers still using this approach, its disadvantages are so significant that we will not consider it further.

The Specifying Paradigm

The specifying approach, commonly known as the *waterfall* model or as *phased refinement*, dictates that software is developed in a series of discrete, successive steps. These steps represent a systematic, sequential approach that include analyzing, designing, coding, testing, and maintaining the system [Pres87]. In contrast to the code-and-fix approach the waterfall model places significant emphasis on documentation [Royc70]. Consequently, the waterfall model can be viewed as an artifact-driven model in which the life cycle phases exist to produce specific artifacts deemed important to the development of the final software system. Artifacts may be of long- or short-term interest. A typical scenario is (i) develop a requirements specification in which all system functionality is specified, (ii) develop a design specification to implement the requirements, (iii) develop code to implement the design with rework occurring as necessary to fix coding problems discovered during testing, and (iv) install and maintain the code [Tani89]. The waterfall model as proposed by Royce is shown in Figure 1. Extensive and rigorous documentation requirements with consistent format and depth of detail are often associated with this approach. Although documentation is important, this seems to place the focus on artifact production rather than on their role of communicating information during system development.

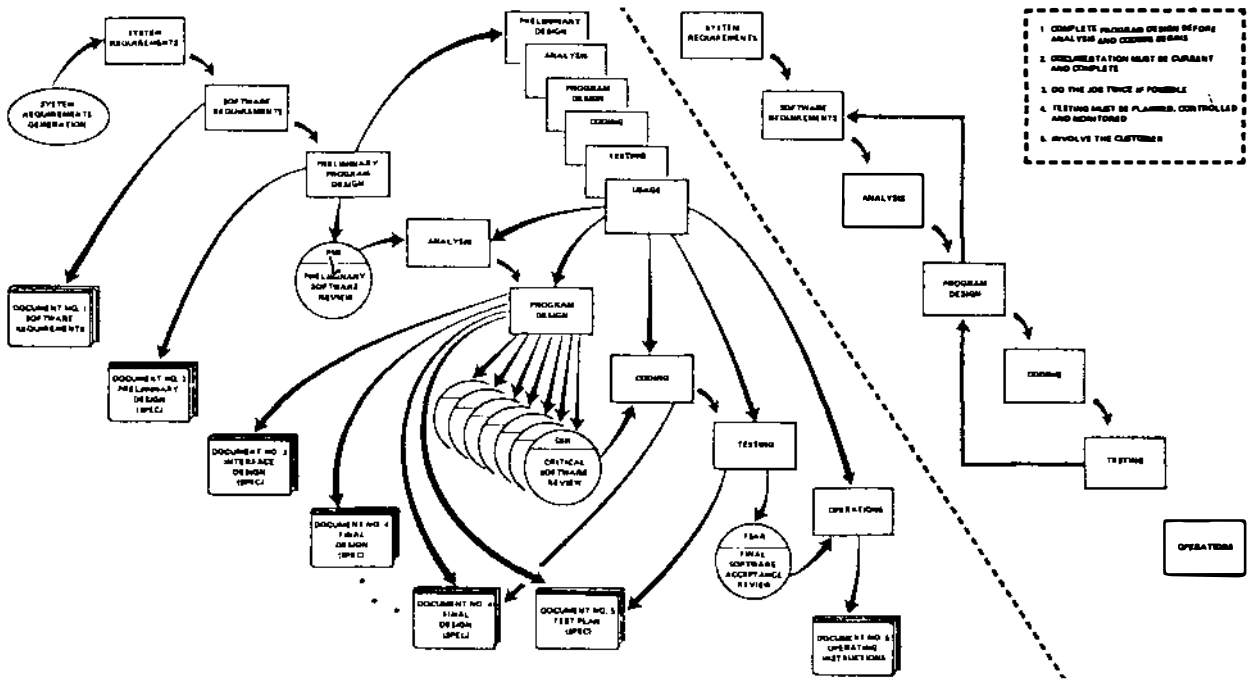


Figure 1. The Waterfall Model [Royc70].

The expectation that the use of specifying will ensure the development of fully elaborated work products at the conclusion of each life cycle phase is a characteristic that many software developers find unnatural and difficult to accomplish. Curtis points out that a major shortcoming of the waterfall model is its failure to treat software development as a problem solving process [Curt87]. He also points out that a model focusing on only the end product of each major activity offers little insight into the actions and events that precede the finished artifact.

Specifying's principle advantages are its recognition of the distinct focus of each phase in which different skills are needed and its emphasis on complete documentation of the work performed. Requirements definition concentrates on ***what*** must be done while design concentrates on ***how*** to do it. Being concerned with delivering a product that satisfies users' needs, the specifying approach attempts to discover errors early by reviewing and analyzing intermediate products prior to initiating the next development phase. Usually, the intermediate products are not conceived with the product's end user in mind, but rather focus on the issues important to the developers. This environment inhibits the effective participation of end users in the review process. Key misconception about what the system is to supposed do may not be uncovered until late in development or after delivery.

The specifying approach is typically criticized as not providing a model of the way people can comfortably work. Creative activities often require a mixture of analysis and synthesis with iteration to refine both the understanding and the solution. Gilb [Gilb88] proposed a variation on the specifying approach that was based on delivering a large number of small, high value increments of capability to the system's end user. He termed this approach *evolutionary delivery*. The greatest benefit of this approach is its emphasis on satisfying user requirements and the opportunity it affords to facilitate early feedback from the user. The risk associated with this approach is that it can degenerate into a code-and-fix process.

The Prototyping Paradigm

Prototyping is a process by which the developers capture critical features in a **model** containing selected aspects of the proposed system [Tani89]. A prototype's purpose is to allow the user to gain experience with the proposed system in order to evaluate whether or not their project expectations are being realized. It is an exploratory process that allows the developers to incrementally discover and refine the requirements. Curtis states: "Managing uncertainty suggests that we reconceive the software life cycle as a learning process rather than a manufacturing process" [Curt87]. The need for dealing with uncertainty requires that techniques be applied to identify and resolve these uncertainties. Prototyping can be an effective technique for addressing uncertainty. Curtis provides another warning about the use of prototyping: "Although prototyping may be useful for answering questions on a piecewise basis during development, it is certainly not the answer at the system level." By itself, prototyping is simply a useful product development technique and is not a development process or paradigm.

This raises the question as to whether prototyping in the absence of a process can offer adequate control to routinely ensure a successful development. There is great risk that the user will not understand the distinction between a prototype and a finished, robust product and will insist that the prototype be delivered and supported.

In his comparison of specifying to prototyping, Boehm found that both prototyping and specifying have advantages which complement each other [Boeh84]. Specifying provided the formalism and documentation necessary for long-term projects, while prototyping enabled the identification and investigation of high-risk issues and provided the flexibility to adapt to the changing perceptions of users' needs.

As a result of his work, Boehm suggested a new paradigm for software development and introduced what is now called the *spiral model* of software development, as shown in Figure 2 [Boeh88]. The spiral model is an iterative risk-driven approach (as opposed to specifying, that is artifact-driven, or prototyping, that is code-driven) that can use both prototyping and specifying techniques.

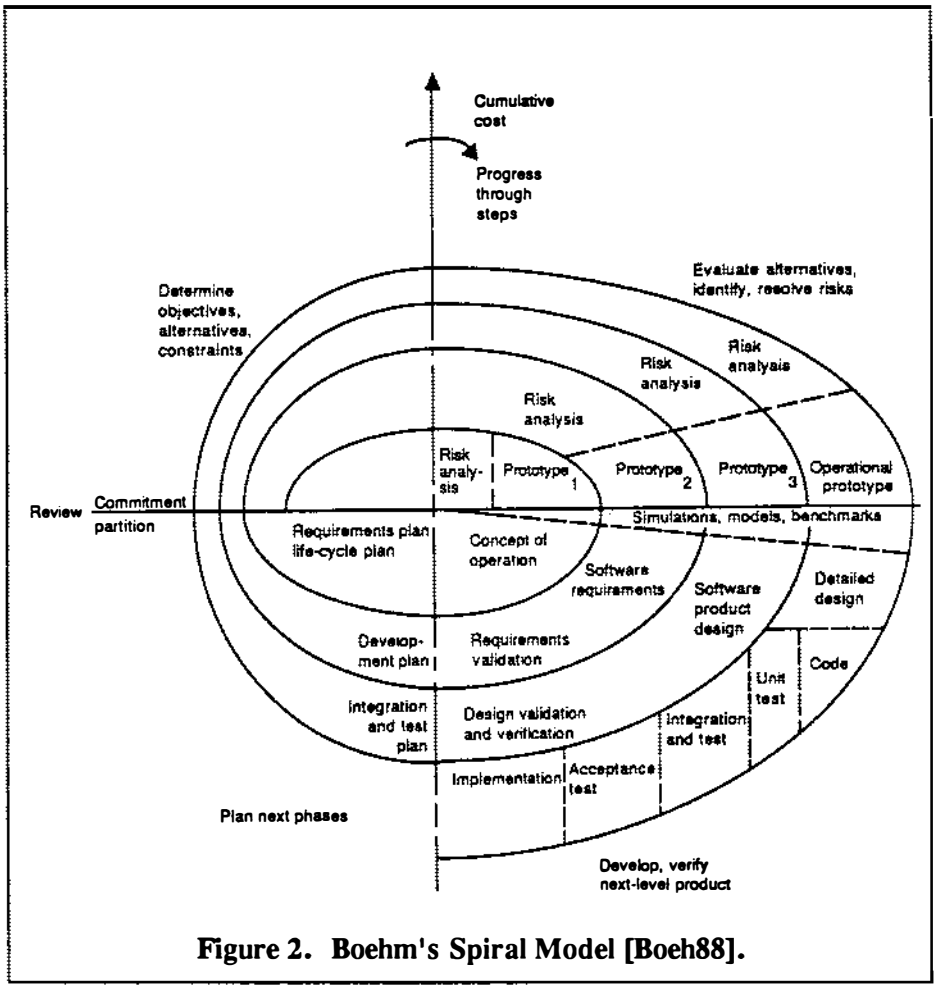


Figure 2. Boehm's Spiral Model [Boeh88].

The spiral model is inherently iterative. Each cycle begins with definition of objectives, alternatives, and constraints. If areas of uncertainty are found that represent significant project risk, then strategies for resolving the sources of risk are formulated. The resolution of the most significant risk drives each iteration.

From an implementation or management perspective, Boehm's presentation of the spiral model is not completely defined. The risk-driven approach places pressure on development teams to correctly identify and manage sources of project risk. It does not inherently provide techniques to identify and manage risk, does not account for people with widely differing experience bases, needs more elaboration of milestones, and needs better techniques for synchronizing schedules [Boeh88]. Some insight into risk management is provided by Boehm in [Boeh91]. He states "The key contribution of software risk management is to create this focus on critical success factors -- and to provide the techniques that let the project deal with them." In essence, the spiral model is still a task-based model although its orientation is substantially different from that of the specifying approach.

EVALUATING LIFE CYCLE MODELS

Evaluating life cycle models is not easy because system development activities are complex processes with many variables and subject to significant statistical variation. In [Curt87], Curtis proposed a scheme for understanding the influences on a project. As projects increase in scope from those that can be accomplished by a single individual, to those requiring a team, and even on to multi-team and larger projects, analysis must consider cognitive issues, as well as issues relating to interactions at the group and organizational levels. While the traditional life cycle models describe how product

information grows and is transformed through a series of artifacts over time, other views are required to analyze the behavioral processes.

Boehm's Study

In 1982, Barry Boehm conducted an experiment to compare the characteristics of products developed via the specification-driven approach to those developed with the prototyping approach [Boeh84]. In his experiment, seven teams developed versions of the COCOMO model for software cost estimation. This was a small-size (2K - 4K lines of code) application software product implementing the same estimation equations but allowing each team to create its own user interface to the model. Four teams used the specifying approach. Three teams used the prototyping approach. The experiment took place as part of a one-quarter (eleven week), first year graduate course in software engineering at UCLA.

The major milestones for the specifying teams were requirements specification, design specification, draft user's manual, acceptance test, final user's manual, and maintenance manual. The major milestones for the prototyping teams were the prototype demo, acceptance test, user's manual, and maintenance manual. The requirements and design specifications were subjected to a thorough review by the instructors. This resulted in a set of problem reports returned to the project teams and discussed in class. The prototypes were exercised by the instructors, who provided similar feedback on errors, suggested modifications, identified missing capabilities, etc.

Boehm and colleagues tested each product and rated it on a scale of 0 to 10 with respect to functionality, robustness, ease of use, and ease of learning. There was also a student subjective rating of the maintainability of the other teams' products.

The main results of Boehm's experiment were: (1) prototyping yielded products with roughly equivalent performance, but with about 40 percent less code and 45 percent less effort; (2) the prototyped products rated somewhat lower on functionality and robustness, but higher on ease of use and ease of learning; and (3) specifying produced more coherent designs and interface specifications which made integration of the software easier.

There were however, some uncontrolled characteristics of Boehm's experiment that may have influenced the results. These problems were team organization, team balancing, team separation, and experimenter bias.

Team Organization: The specifying teams were staffed entirely with students who had expressed a preference for the specifying approach, and similarly for the prototyping teams. Within each team, team members were free to organize in whatever way suited them, but most adopted a democratic consensus based arrangement. This paradigm preference and management style are non-representative of real world product developments where organizational policy or culture determines the development approach.

Team Balancing: The prototyping teams were smaller and had more experience that the specifying teams in both general programming ability and with the language used for development, Pascal. Table 1 details the team organization and balancing. Differences seen in the product could be caused by the smaller average team size of the prototyping groups and by the widely differing Pascal and programming experience between the prototyping and specifying subjects.

Team Separation: Requirements and design reviews and prototype demonstrations were conducted in front of the entire class. Prototypers particularly benefited from insight gained during specifier's reviews. To a lesser extent some specifiers may have benefitted from seeing the prototype demonstration prior to completing their product's design. Preventing cross-fertilization between teams

was not considered as important as the anticipated gain in teaching effectiveness that in-class reviews and demonstrations would provide.

Team Averages	No. in Team	Prog. Exp. *	Pascal Exp. *	Unix Exp. *	GPA
Specifying Teams	2.75	36	7	4.5	3.37
Prototyping Teams	2.33	53	18	2.3	3.27

* Time in Months

Table 1. Team Balancing in Boehm's Study

Experimenter Bias: The experiment's authors rated each product in terms of functionality, robustness, ease of use, and ease of learning. Since the authors knew which paradigm had been used to develop each product, this knowledge could have lead to an unconscious bias toward or against a particular approach. Rather than evaluating the projects themselves, they should have used an independent group of experts to rate the products, without any knowledge of the paradigm employed.

Despite these problems, Boehm's experimental results appear reasonable and his conclusions appropriate. He concluded that both the specification-based and prototyping approaches have strength, and in fact complement each other. Specifying provides the formalism and documentation necessary for large, long-term projects. Prototyping enables the identification of high-risk issues, provides users with early system experience, and has the flexibility necessary to accommodate changing user requirements. This led to his conclusion that what was needed was some project specific mix of specifying and prototyping arrived at by risk management. Risk management dictates that software projects should develop, maintain, and follow plans that identify potential high risk issues, establish plans for their resolution, and emphasize risk resolution in product status reviews.

Unresolved Issues: Prototyping seems to offer advantages, such as facilitating early identification of high risk issues and the flexibility to adapt to changing perceptions of the user's needs. The question is: When and where should the differing software development approaches be used? Although prototyping appears to have advantages, it is not really known if it offers the degree of process control that specifying provides. Furthermore, the characteristics of products appropriate for prototype development is not well understood. Finally Boehm saw significantly less code and effort by the prototypers in his experiment, but it is not known if this is a general characteristic to be expected from the approach or whether it might be due to differences in documentation requirements and experience levels. It is not clear what the differences are between the approaches, and which activities are responsible for those differences.

Aside from Boehm's comparative study which showed significantly higher productivity and better ease of use for systems developed using the prototyping approach [Boeh84], there have been no published accounts of controlled studies comparing development approaches and evaluating the effects of the development paradigm on software product quality.

OUR EXPERIMENTS

The differences in quality characteristics of the products developed under different paradigms is not well understood. The question is, when and where should each approach be used, and what are the effects on the end products? This is the question that motivated our series of experiments.

Experimental Procedures: In order to study these issues in a more controlled environment, we conducted a series of three individual experiments which compared the traditional specifying approach to a spiral-based prototyping approach. Each experiment involved two balanced teams of experienced student programmers enrolled in a senior-level, one semester (16 week), software engineering practicum. The teams were given identical requirements from an independent customer who needed a system developed. One team was selected to develop the product using a specifying approach while the other team used a spiral-prototyping approach. Team composition was controlled by the course instructors in order to provide roughly equivalent capabilities in both teams. Teams were isolated to avoid cross-fertilization of ideas.

The Specifying Life Cycle: The major features of the specifying life cycle used in our experiments are shown in Figure 3. The specifying teams produced a requirements specification document, design specification document, final code and test materials, and a user's manual. The requirements and design documents were produced following the IEEE Standard 830 and IEEE Standard 1016 respectively. The teams were expected to update their documentation so that at the end of the project, it reflected all requirements and design changes implemented subsequent to initial document preparation. The teams conducted a system design review that focused on the system architecture, user interface, and major files and data structures. A final product installation and demonstration concluded the project.

I. Project Initiation
   1. Project Planning
      a. Schedule
      b. Team assignments

II. Requirements Definition
   1. Customer Interviews
   2. Requirement Document
      a. Usage scenario
      b. Data flow diagrams
      c. Data dictionary

III. Design
   1. System Architecture
      a. Structure chart
      b. User interface design
      c. File & data structure design
   2. System Design Review
   3. Detailed Design
      a. Module pseudo code
   4. Design Document

IV. Coding & Testing
   1. Coding
   2. Unit Testing
   3. Integration

V. Implementation
   1. User's Manual
   2. Documentation Update
   3. Installation & Demonstration

Figure 3. Specifying Life Cycle

The Spiral-Prototyping Life Cycle: The prototyping teams created three prototypes, then final code and test materials, a user's manual, and a maintenance document. A spiral-based prototyping model was used to support an incremental development approach. Each cycle consisted of four phases: (1) planning/analysis phase, (2) specify/prototype phase, (3) test/review phase, and (4) analysis/replanning phase. The activities occurring during each phase may be seen in Figure 4. Each cycle was initiated by consideration of the most significant risk item facing the development and each cycle was concluded by a prototype demonstration and a review. The focus of this review was to assess the strengths and weaknesses of the prototype, how adequately it had resolved the high risk area, and to identify the content of the next prototype. A final product installation and demonstration concluded the project.

Team Balancing: Balanced teams were assembled to minimize the effect of variations in capabilities on the experiment's results. Team balancing data are shown in Table 2. Balancing factors included: number of computer science courses completed, experience with mainframe and minicomputers, experience with microcomputers, number of languages known, work experience, implementation language experience, grade point average, total number of credits, and a self-determined subjective rating of their implementation language proficiency (Rate 1) and general programming abilities (Rate 2). For Experiments 1 and 2 implementation language experience and proficiency were assessed with respect to the Pascal language while in Experiment 3 they referred to the "C" programming language.

I. Planning/Analysis Phase
   1. Statement of the objectives
   2. Known constraints
      a. System
      b. Time
      c. Other
   3. Alternatives
      a. Feasible
      b. Other
      c. Model descriptions
   4. Potential problems
      a. Problem statements/ possible resolutions

II. Specifications/Prototype Phase
   1. Prototype minispecs
      a. Data flow diagrams
      b. Structure chart
      c. Module description
   2. Prototype code
      a. Prototype driver
      b. Modules & stubs

III. Testing/Review Phase
   1. Minimal test set generation
   2. Test execution
   3. Problems encountered
      a. Problem statement
      b. Actual and/or proposed solution(s)

IV. Analysis/Replanning Phase
   1. V&V Checklist
   2. Plan for next cycle
      a. Statement of goals
      b. Team member commitments

Figure 4. A Modified Spiral-Prototyping Model .

Team Separation: In Experiments 1 and 2 a customer proxy was used to distance the real customer from the ongoing experiment and to minimize the interaction between the two independent teams. Course instructors acted as the customer proxies. The customer provided the requirements and answered questions directly to both teams. Information supplied to one team was also supplied to the other team if it related to clarification of, or change in requirements. The customer was not involved in any design/code reviews or prototype evaluations. For detailed review and testing both teams dealt directly with the course instructors (experimenters). The customer proxies were used so that the real customer's perception of the system being developed was not altered by its ongoing development. Further, we didn't want the customer to "cross-fertilize" the two development projects by unwittingly passing information between teams. Although from an end-product perspective it certainly would have been advantageous to have the real customer involved in the development, it was more important to keep a barrier between the teams.

Team Averages	No. in Team	GPA	Total Credits	CS Classes	Work Exp.*	No. of Lang.	Impl. Lang. Exp. *	Mains & Minis	Micros	Rate 1 Lang. Prof.	Rate 2 Prog. Ability
Experiment 1: FSM Simulator											
Specifying Team	5	2.62	122.2	11.8	9.0	5.0	26.0	2.6	3.6	4.2	4.4
Prototyping Team	5	2.72	120.0	12.8	5.4	6.0	31.2	3.2	2.8	4.8	4.4
Experiment 2: TVB											
Specifying Team	5	2.41	108.4	9.4	4.8	3.8	13.2	2.6	1.6	3.8	3.4
Prototyping Team	4	3.36	87.5	11.5	1.0	2.5	23.2	2.0	1.25	3.5	3.25
Experiment 3: DETECH											
Specifying Team	3	2.69	117.7	14.6	5.0	5.0	28.0	3.0	2.0	3.6	3.6
Prototyping Team	3	3.18	128.0	12.6	4.3	6.0	27.3	3.0	3.33	4.6	3.5

* Time in Months

Table 2. Team Balancing Data

Experimenter Bias: The final product's quality was measured by customer satisfaction and reported errors. In addition to the customer's evaluation, data were collected to measure code complexity through a battery of complexity metrics. The separation between the development teams and the customer facilitated a more objective final evaluation of each product because the customer was not intimately familiar with the implementation details. The complexity data are objective measures, independent of the experimenters or customer. After the end products were delivered, the customer was required to test the products, complete an evaluation form, and produce an error report. Because of this procedure we believe that we have effectively eliminated experimenter bias for our experiments.

Measures: Data were collected to gauge team effort, code complexity, and software quality. Effort was measured by hours worked on the project, software complexity was measured with a battery of complexity metrics, and software quality was measured by customer satisfaction and reported errors.

Effort data were collected weekly from team members during a scheduled team meeting. Team members were required to complete forms that asked them to log their time expended for each project activity they had performed.

Complexity data were calculated with a metric analyzer that was applied to the final code from each development project. These metrics were reported on a module-by-module basis and were then totaled for the entire system.

In Experiments 1 and 2 the subjective customer evaluation consisted of several five-point, forced-choice positive statements, with responses ranging from 1-strongly disagree to 5-strongly agree. In each experiment the customer tested and evaluated both products. In Experiment 3 the customer was asked to evaluate each team in several areas relating to the effectiveness of their development approach.

During customer's testing, reported errors were recorded for both projects and later categorized according to errors of omission and errors of commission. Errors of omission are the results of incorrect or missing requirements. Errors of commission are errors for which requirements were correct but the implementation was flawed.

Experiment 1: The product to be developed was a visual simulator for Finite State Machines (FSM) as described by Jagielski in the *ACM SIGCSE Bulletin* [Jagi88]. The system was to allow the user to create and display a FSM state transition diagram on the computer screen, accept an input sentence from the user, and check the sentence for being in the FSM's language. As it checked the sentence, the nodes and arcs in the state transition diagram were to change color to show the path taken. Upon checking the entire sentence, a message was to be displayed as to whether the sentence was accepted by the FSM or not.

Options for saving and loading FSMs to/from disk and having keyboard or file entry during simulation were required. Additional requirements stipulate that the program be developed in Pascal, that it must run on an IBM compatible PC with EGA graphics, and that it must be able to print the FSM state transition diagram.

Experiment 2: The product to be developed was a system used to convert document files developed with the TEX publishing package to files fitting a VENTURA format. This system was termed the TEX-to-VENTURA Bridge (TVB). TEX and VENTURA are independent commercial desk top publishing packages that produce documents containing typesetting codes. It was to first prompt the user as to whether instructions were needed, and display them if necessary. The user was then required to enter the file name of the TEX file to convert and the VENTURA file to create.

Two windows were to be used to display the TEX to VENTURA conversion process, with a user controllable scrolling speed. One window displayed the TEX input file, the other window showed the VENTURA file as it was being formatted. If the system found a TEX code that it could not convert, the user would be prompted to enter either a null or substitute VENTURA code. An external ASCII table that could be modified with a text editor was required to define actions between two equivalent TEX and VENTURA codes. Other requirements mandated that the system be developed in Pascal, and must run on an IBM or compatible PC.

Experiment 3: The product to be developed was a code similarity measurement tool used to evaluate suspected illicit program derivations. The program, DETECH, views a source program file as a string of words from which certain key words are extracted and counted. Similarity between programs is determined by assessing the programs in three orthogonal dimensions that measure program structure, complexity, and style. DETECH assesses and reduces each of the three analyses to a single fixed measure of similarity.

The program was to accept a file name for an "original" program and a file name for a suspected derivation. It was to read each program and compute the similarity measures for each, and then report

the accumulated and calculated data for both programs. The program was to be written in C and was to execute on the Apollo computer system.

RESULTS

In the following paragraphs we present a discussion of the results observed in our experiments. The data are summarized and presented in Table 3.

Product Complexity Comparisons: For the FSM Simulators (Experiment 1), both products were of similar total size. Overall, the prototyped product contained 10% fewer lines of code. In a detailed evaluation of the code, we found that the prototyped product contained instances of functionally redundant code that could have been implemented as utility modules. This would have further reduced the program's size.

Despite the fact that these products were of similar total size, the prototyped product contained 57 modules, while the specified product only contained 34. The average V_g (McCabe's Cyclomatic Control Flow Complexity) for the specified product was 12.7 compared to 7.9 for the prototyped product.

The TVB products (Experiment 2) were of considerably different size. The specified product contained 1650 delivered source lines as opposed to 4771 delivered source lines in the prototyped product. This large discrepancy occurred because the prototyping team used cut-and-paste editing to replicate a large amount of code with small editing changes to account for special cases. This generated a considerable amount of functionally redundant code, and can be viewed as a characteristic of their programming style. Although the metrics indicated the prototyped product contained substantially more lines of code and tokens, the per module average is slightly lower in the prototyped product. The prototyped product contained 109 modules, while the specified product only 34. Despite the fact that these products were of very different sizes, the result is that the specified product's modules were on average more complex than those of the prototyped product. The specified product had an average V_g of 9.94 compared to 6.8 for the prototyped product.

In both the FSM Simulator and the TVB products, the nesting of control structures was lower in the prototyped products, indicating that they were constructed from modules of lower average complexity. We also observed that the average Halstead metrics N1 and N2, are substantially lower in the prototyped implementations, again suggesting the existence of more compact modules. Although in these two product, the prototyped implementations contained more modules, the average number of arguments per module are about the same. The increase in modules did not seem to adversely affect the intermodule communication.

Although only a subset of the metrics were available from the C language implementations of DETECH (Experiment 3), the same patterns were observed. Again, the prototyping team showed a tendency to produce a system with more modules and a smaller average module size. The distinction between the two products was particularly evident with respect to V_g. In the specified product it averaged 10.1 compared to only 4.8 in the prototyped product.

The metric averages support the conclusion that the prototyping approach results in smaller and less complex modules when compared to the same product developed using the specifying approach. This was observed in all three experiments.

Development Effort: Effort profiles for each experiment and each team are shown in Figure 5. In Experiment 1 the specifying team logged a total of 637 hours to complete the project compared to 478 hour logged by the prototyping team. Observable in the figure is the tendency for effort to be driven by approaching deadlines. Effort peaks near the point where a deliverable, a document or prototype, is due and tends to decrease sharply after the milestone is completed.

		No. of Modules	DSL	LOC	TOK	ARG	COM	V_g	NST	n1	N1	n2	N2
Experiment 1: FSM Simulator													
Specifying Team	Total	34	2249	1977	14966	84	938	431	165	607	6517	1079	4670
	Module Avg.		66.2	58.2	441.1	2.5	27.6	12.7	4.9	17.9	191.7	31.7	137.4
Prototyping Team	Total	57	2172	1686	13423	125	2069	449	238	930	5479	1273	4101
	Module Avg.		38.1	29.6	235.5	2.2	36.3	7.9	4.2	16.3	96.1	22.3	72.0
Experiment 2: TVB													
Specifying Team	Total	34	1650	1462	8790	63	275	338	177	583	4092	571	2399
	Module Avg.		48.5	43	259	1.85	8.09	9.94	5.21	17.2	120.4	16.8	70.6
Prototyping Team	Total	109	4771	4149	26588	225	2745	742	282	1522	12205	1920	8183
	Module Avg.		43.8	38.1	244	2.06	25.2	6.8	2.6	14	112	17.6	75.1
		No. of Modules	*	LOC	*	*	*	V_g	*	n1	N1	n2	N2
Experiment 3: DETECH													
Specifying Team	Total	48		4416				484		680	5828	788	3574
	Module Avg.			92.0				10.1		14.2	121.4	16.4	74.5
Prototyping Team	Total	62		3794				300		788	4096	826	2075
	Module Avg.			61.2				4.8		12.7	66.1	13.3	33.5

* Not available for "C" language implementation

DSL - Delivered Source Lines (excluding comments)
LOC - Lines of Code (executable lines)
TOK - Tokens
ARG - Arguments
COM - Comments

V_g - McCabe's Measure
NST - Level of Nesting
n1 - Halstead's Unique Operators
N1 - Halstead's Total Operators
n2 - Halstead's Unique Operands
N2 - Halstead's Total Operands

Table 3. End-Product Complexity Comparisons

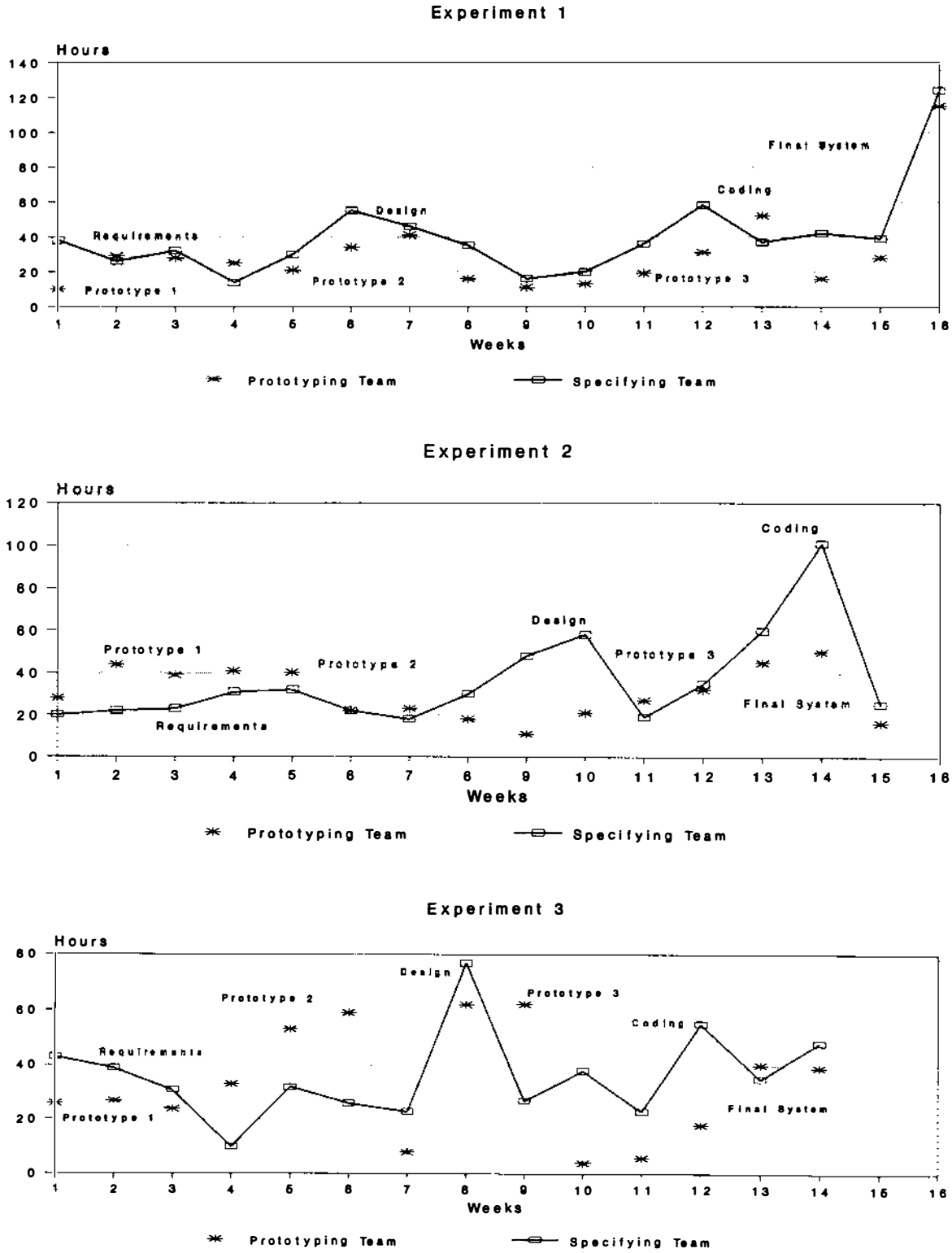


Figure 5. Weekly Effort Data for Each Experiment

For Experiment 2 the deadline effect is again clearly visible for the specifying team but less noticeable for the prototyping team. In this project the specifying team logged a total of 568 hours compared to a total of 455 hours for the prototyping team.

In Experiment 3 the specifying team once again required more effort to complete the project, 507 hours compared to 461 hours for the prototyping team. Deadline effects are still clearly visible for both teams.

Customer Evaluation: On the 20 question, 100 point subjective customer evaluation, the prototyping team's FSM Simulator was rated a score of 90 compared to a 66 for the specifying team's product. Both a pairwise t-test ($T=4.7$, d.f. $= 19$, $p < 0.001$) and a Wilcoxin signed-rank test ($W_+ = 4$, $N = 15$, $p < 0.01$) indicate this difference is significant.

On a 15 question, 75 point subjective customer evaluation, the prototyping team's version of the TVB product received a score of 68 compared to the specifier's score of 54. Both a pairwise t-test ($T = 4.52$, d.f. $= 14$, $p < 0.001$) and a Wilcoxin signed-rank test ($W_+ = 0$, $N = 11$, $p < 0.005$) indicate the difference is significant.

The customer for the DETECH produced expressed a preference for the specifying team's product primarily due to several implementation errors in the counting strategy in the prototyper's product. However, he preferred for the user interface of the prototyper's product. On a 5 question, 25 point subjective evaluation the customer rated both teams equally with a score of 22. The quality of the prototyper's code was rated higher, while the specifiers were rated higher on overall satisfaction of project requirements.

Reported Errors: Table 4 presents a summary of the customer reported errors found in each product. Errors are classified as either errors of omission or errors of commission. Errors of omission are the result of incorrect or missing requirements. Errors of commission are errors for which requirements were correct but the implementation was flawed.

For the FSM Simulator products, the customer reported eight errors in the specified product and four errors in the prototyped product. While both products had two errors reported as errors of commission, the specified and prototyped products contained six and two error of omission respectively. Significant problems were reported in the user interface of the specified product.

For the TVB products, there were six flaws in the specified product and three flaws in the prototyped product. While both products had three errors of commission, the specified product also contained three errors of omission while the prototyped product contained none. Significant flaws in the specified product were

	Errors of Commission	Errors of Omission
Experiment 1: FSM Simulator		
Specifying Team	2	6
Prototyping Team	2	2
Experiment 2: TVB		
Specifying Team	3	3
Prototyping Team	3	0
Experiment 3: DETECH		
Specifying Team	3	3
Prototyping Team	5	0

Table 4. Reported Errors for Each Project

related to an overly simplistic user interface and unexpected program behavior.

For the DETECH product, total errors were about the same for each team. However, the prototyping team's errors were all classed as errors of commission while the specifiers errors were evenly split between errors of commission and errors of omission. The prototyper's errors were related to inaccurate counting. The specifiers failed to implement some requirements, including getting the product to execute on the target computer system.

CONCLUSIONS AND RECOMMENDATIONS

We have established a mechanism for investigating software development paradigms that ensures comparability of data collected from independent software development teams. Using this mechanism we have shown that when comparing prototyping to specifying, the prototyping process facilitated production of a larger numbers of smaller, less complex modules. These modules took less effort to develop, and contained fewer errors as a result of software reuse through the evolution of the prototypes. We were particularly encouraged to see that prototyped products consistently contained fewer errors of omission.

There was also a reduced deadline effect in the prototyping projects. Effort curves were smoother than those of the specified projects. Teams using prototyping not only completed their projects with less effort, but that effort also seemed to be more evenly distributed throughout the project.

The larger module size and V_g observed in the specified product led us to evaluate the code itself. We found segments of unnecessarily duplicated code in multiple modules. Based on our observations we can postulate that developers using specifying may have had a tendency to stick to the specified design past the point when it should have been revised. The design probably did not decompose the system into an adequate number of modules or failed to identify operations that could have been made into utility modules. Some of the reluctance to create additional modules during the coding phase may have been due to the mandate to modify formal requirements and design documents so that they accurately reflected the finished product. The reluctance to change was manifested in modules containing additional code to implement details unforeseen at design time.

Interesting, we also observed that developers using prototyping unnecessarily duplicated code. We can postulate that they are often not looking past the immediate prototype when implementing their systems. It was more convenient to copy existing code and implement minor changes than it was to create a general purpose module. The lack of design documentation in the prototype developments may also have contributed to this tendency if team members, in the absence of detailed knowledge about other portions of the system, independently developed functionally equivalent code to support their assigned area. Unless there is intent to re-engineer the system at a later time, care should be taken to evaluate design choices in the context of how they will affect the final system. This nearsightedness can lead to systems being implemented with inefficient designs which are difficult to maintain and can lead to functionally redundant code segments.

Although we have shown that prototyping reduces errors of omission, if not careful, prototypers may lose sight of the original requirements as their systems evolve and may run a risk of failing to deliver a finished product. Customers and developers using prototyping must take care to examine each new requirement as to how it supports the original requirements, what implications it has on the system architecture, and revise the system goals as necessary. They should not let the prototyping take on a momentum of its own.

Configuration management techniques are needed to keep prototype histories and project baselines intact in order to ensure that a code-and-fix strategy does not emerge. As part of this baseline,

prototypes should be accompanied by requirements specifications and design materials which document the important architectural decisions. Evolving customer requirements should also be documented as part of this baseline. Prototypes themselves do not adequately express the requirements of a system. Developers using prototyping need additional material to validate the existence of each prototype component as well as mechanisms to prevent them from simply finishing the project and then tailoring the final requirements to coincide with the final delivered system.

The modified spiral model was effective in encouraging evaluation of product constraints and alternatives. It highlighted risk item resolution by having the development team evaluate problems and possible problem resolutions during each cycle. It helped give the process some structure that is normally lacking in a prototyping environment, with the desired result that the developers were not free to just start coding. It also gave the (proxy) customer some early experience with the system and some opportunities for affecting the direction of the ongoing development. The modified spiral model template gave both the developers and the customer a good plan to follow as well as the needed confidence that they were reaching meaningful project milestones.

REFERENCES

[Boeh84] B. W. Boehm, T. E. Gray, and T. Seewaldt, "Prototyping Versus Specifying: A Multiproject Experiment," *IEEE Transactions on Software Engineering*, Vol SE-10(3), 1984, pp. 290-302.

[Boeh88] B. W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, May 1988, pp. 61-72.

[Boeh91] B. W. Boehm, "Software Risk Management: Principles and Practices," *IEEE Software*, January 1991, pp. 32-41.

[Curt87] Bill Curtis, Herb Krasner, Vincent Shen, and Neil Iscoe, "On Building Software Process Models Under the Lamppost," *Proceedings of the 9th International Conference on Software Engineering*, March 30-April 2, 1987, pp. 96-103.

[Gilb88] Tom Gilb, *Principles of Software Engineering Management*, Addison-Wesley, 1988.

[Hagi88] R. Jagielski, "Visual Simulation of Finite State Machines," *ACM SIGCSE Bulletin*, Vol 20(4), December 1988, pp. 38-40.

[Pres87] Roger Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 1987.

[Royc70] Winston W. Royce, "Managing the Development of Large Software Systems," *Proceedings of IEEE WESCON*, August 1970, pp. 1-9.

[Tani89] M. M. Tanik and R. Yeh, "Rapid Prototyping in Software Development," *IEEE Computer*, May 1989, pp. 9-10.

Rapid Prototyping and Software Quality: Lessons From Industry

V. Scott Gordon James M. Bieman

Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA
(303) 491-7096
gordons@cs.colostate.edu, bieman@cs.colostate.edu

Abstract

Empirical data is required to determine the effect of rapid prototyping on software quality. In this paper, we examine 24 published and unpublished case studies, in order to report "real world" experiences. We analyze the case studies to identify common observations, unique events, and opinions. We develop guidelines to help software developers use rapid prototyping in such a manner as to maximize product quality and avoid common pitfalls.

The Authors

- V. Scott Gordon is a Ph.D. student in Computer Science at Colorado State University. He received his B.S. and M.S. degrees in Computer Science at California State University, Sacramento, and has experience as a software developer working for TRW Defense Systems Group. Mr. Gordon's research interests include rapid prototyping, genetic algorithms, and neural networks. He is a member of Upsilon Pi Epsilon and received the 1989 Distinguished Alumni Award from the CSUS School of Engineering and Computer Science.

- James M. Bieman is an Associate Professor in the Computer Science Department at Colorado State University. Dr. Bieman's research is focused on software structure and measurement, testing strategies, software specification and prototyping. He has published in journals including the *Software Engineering Journal*, *Journal of Systems and Software*, and *Computer Languages*, and conferences including COMPSAC and the ACM Testing, Analysis, and Verification Symposium (TAV). He recently served on the Highly Reliable Software Peer Review Committee at NASA Langley Research Center. Dr. Bieman is coordinating the establishment of a Reliable Software Laboratory at Colorado State University.

1 Introduction

Prototyping affords both the engineer and the user a chance to "test drive" software to ensure that it is, in fact, what the user needs. Also, the engineer gains understanding of the technical demands upon, and the consequent feasibility of, a proposed system. *Prototyping* is the process of developing a trial version of a system (a *prototype*) or its components in order to clarify the requirements of the system or to reveal critical design considerations. The use of prototyping may be an effective technique for correcting weaknesses of the traditional "waterfall" software development life cycle by educating the engineers and users [Har87].

Does the use of rapid prototyping techniques really improve the quality of software products? The relationship between development practices and quality must be determined empirically. Our objective is to learn how to improve the quality of software developed via rapid prototyping by drawing on the experiences of documented "real world" cases.

In this paper, we investigate the effect of rapid prototyping on software quality by examining both published and unpublished case studies. These case studies report on the actual use of rapid prototyping in developing military, commercial, and system applications. We analyze the case studies to identify common experiences, unique events, and opinions. We develop some guidelines to help software developers use rapid prototyping in such a manner as to maximize product quality and avoid common pitfalls.

The nomenclature regarding prototyping varies [Pat83]; we use the following definitions: *Rapid prototyping* is prototyping activity which occurs early in the software development life cycle. Since, in this paper, we are only considering early prototyping, we use the terms "prototyping" and "rapid prototyping" interchangeably. *Throw-away* prototyping requires that the prototype be discarded and not used in the delivered product. Conversely, with *keep-it* or *evolutionary* prototyping, all, or part, of the prototype is retained in the final product. The traditional "waterfall" method is also called the *specification approach*. Often prototyping is an iterative process, involving a cyclic multi-stage design/modify/review procedure. This procedure terminates either when sufficient experience has been gained from developing the prototype (in the case of throw-away prototyping), or when the final system is complete (in the case of evolutionary prototyping). Although there is some overlap between rapid prototyping and executable specifications [LB89], we concentrate here solely on rapid prototyping. We generally follow the taxonomy outlined in [Rat88].

This paper is organized as follows. In Section 2 we describe our research methods. Section 3 describes seven effects of prototyping on software quality. Section 4 discusses common beliefs regarding rapid prototyping on quality. We sort out conflicting recommendations concerning four frequently debated questions regarding proper prototyping methods. In Section 5, we describe potential pitfalls associated with prototyping that are revealed by the case studies. We suggest some simple steps to avoid the pitfalls. We summarize our results in Section 6. The References include brief descriptions of each case study as well as general works on prototyping.

2 Nature of Study

For this study, we collected actual case study reports for analysis. Our information is from several available and appropriate sources. These sources include published reports and unpublished communications.

Although many research papers concerning rapid prototyping have been published [LB89, Mit82, Rat88], few papers report on actual real-world experience. We located 17 published reports representing 16 case studies. The earliest case study is from 1979, while most are from

the mid-to-late 1980's (industry use of rapid prototyping appears to be a relatively recent phenomenon). Accounts come from a variety of sources including *Communications of the ACM*, *ACM Software Engineering Notes*, *IEEE Computer*, *Datamation*, *Software Practice and Experience*, *IEEE Transactions on Software Engineering*, and several conference proceedings.

To supplement these published reports, we found additional reports through the internet news service. Through this network search, we have unpublished reports and personal communications from seven individuals closely associated with rapid prototyping. We also include two papers which analyze other rapid prototyping cases. Thus, we have 24 sources of case study information. The sources represent a variety of organizations: AT&T, General Electric, RAND, MITRE, Martin Marietta, Los Alamos, ROME Air Development Center, Hughes, U.S.West, data processing centers, government divisions, and others. Nine of the sources are projects conducted at Universities, but only two of these are student projects. Seven of the sources describe military projects.

The data is not without bias. For example, in 22 of the 24 individual case studies, rapid prototyping is deemed a success. This encouraging result must be tempered by the observation that failures are seldom reported. Some of the sources, however, do address intermediate difficulties encountered and perceived disadvantages of rapid prototyping. Another possible bias occurs in the two sources involving student projects. Boehm describes the inherent bias: "Nothing succeeds like motivation [when] 20 percent of your grade will depend on how much others want to maintain your product" [BGS84]. Finally, four of the sources describe projects which involve no customer *per se*; the goal of these projects is the development of a system to be used by the developers. We do not draw strong conclusions regarding clarity of requirements or successful analysis of user needs when a project does not involve a separate user.

In our analysis, we examine the conclusions made in the case studies with a focus on the impact of rapid prototyping on software quality. Case studies varied in degree of rigor. Three sources observe multiple projects and present conclusions based on quantitative measurements of the results [Ala84, BGS84, CB85]. Others offer subjective conclusions and suggestions acquired from personal experience in a particular project. Some of the studies include a minimal amount of quantitative measurement interspersed with subjective judgement. We emphasize conclusions that were reached by multiple sources independently.

3 Software Quality Effects

The sources describe the effect of prototyping on several aspects of software quality. Most of the described effects are positive.

Improved ease of use

Twelve sources indicate that products developed via prototyping are easier to use. Users have an opportunity to interact with the prototype, and give direct feedback to designers. For example, Gomaa [Gom83] describes how, in some cases, users are not sure that they want certain functions implemented until they actually can try them. Users may also find certain features or terminology confusing. Also, the need for certain features may not be apparent until the system is actually exercised. In Zelkowitz [Zel80], the author "soon tired of retyping in definitions for each ... run", pointing to a need for the capability to store function definitions.

Eleven sources observe more enthusiastic user participation in the early stages of requirements definition. Users are more comfortable reacting to a prototype than reading a "boring" [GS81,

GHPS79] abstract written specification. No sources indicate that software produced via rapid prototyping was more difficult to use.

Better match with user needs

Twelve sources confirm Brooks' famous maxim, "plan to throw one away; you will, anyhow" [Bro75]. Our interpretation is that software developed without a prototype is less likely to meet actual user needs, and be discarded. Sources indicate that prototyping tends to help ensure that the first implementation (after the prototype) will meet users needs: "Omissions of function are often difficult for the user to recognize in formal specifications" [SJ82]. "Prototyping helps ensure that the nucleus of a system is right before the expenditure of resources for development of the entire system" [Ala84]. Only one source indicates that the software produced did not meet users' needs [CB85].

Effect on performance

Three sources suggest that inferior performance is a possible pitfall, especially with evolutionary prototyping.

> "The emphasis in rapid prototyping is typically on proof of concepts rather than performance. However, a programmer should consider performance as early as possible *if the prototype is to evolve into the final system*" [GCG+89].

Developers who intend to use a significant portion of the prototype in the final system need to take steps to ensure adequate system performance (see Section 5).

Effect on design quality

Sources report that keep-it prototyping sometimes results in a system with a less coherent design and more difficult integration. This negative effect can contribute to project failure [CB85] and can impact successful projects [BGS84].

On the other hand, the multi-stage design/modify/review process can result in significantly better overall design. Ford and Marlin state that prototyping "allows early assessment of the efficiency of techniques required to implement specific features" [FM82]. Overall, three sources cite improvement in design quality [Hek87, CB85, FM82], while three sources observe deterioration [BGS84, CB85, Tam82]. See Section 5 for specific recommendations.

Effect on maintainability

Maintainability effects vary. The case studies do not support the belief that keep-it prototyping results in software that is impossible to maintain. Five sources cite improvement [Hek87, BGS84, CB85, A4, Tae91], while only two sources note a reduction in maintainability [GCG+89, CB85]. Hekmatpour describes experiences of maintaining a large system developed via evolutionary prototyping: "The ease with which these modifications were made ... confirms the contention that prototyping can lead to maintainable products" [Hek87]. In analyzing this particular project, we infer that the high degree of modularity required for *successful* evolutionary prototyping can lead to easily maintainable code. Connell and Brice observe that "the modular style of rapid prototype development leads to reusable and replaceable functional modules" [CB85].

There are also indirect reductions in maintenance costs owing to the greater likelihood that user needs will be met *the first time*, reducing the "maintenance" associated with changing requirements [Tae91]. However, some pitfalls should be avoided (see Section 5).

Code length

Three sources report that prototyping results in shorter final programs [AB90, Hek87, BGS84]. No sources report an increase in code length. Boehm's explanation is that prototyping encourages a "higher threshold for incorporating marginally useful features" [BGS84]. Prototyping places a quicker burden of implementation on the designer, and reduces the *talk is cheap* effect of unnecessary promises which are implemented as unnecessary code. Thus prototyping may have a *streamlining* effect. That is, less critical features are less likely to be included in the final system, since the prototype reveals which features are essential. Boehm observed a 40% reduction in source code. Code size may also be reduced when a special-purpose prototyping language is employed [AB90], because features specifically useful for a particular project are typically codified in the language itself and thus do not need to be coded explicitly.

Fewer bells and whistles

None of the sources report an increase in features in the systems due to prototyping. Several sources report discarding some features in favor of others, and three cases specifically cite a reduction in the total number of software features [BGS84, CB85] ([CB85] references two case studies).

This effect is perhaps counter-intuitive. One might expect that the prototyping paradigm gives the end user a license to demand more and more functionality. Actually, Boehm observes that it is more likely that the process will cause critical components to be stressed, and non-critical features to be suppressed [BGS84]. Connell and Brice observe a reduction in features in both successful and unsuccessful cases [CB85].

4 Common Questions

The rapid prototyping literature reveals a number of controversial topics. We describe common questions which are relevant to software quality, and summarize the often conflicting recommendations of our sources.

Should the prototype be kept, or thrown away?

Many engineers are adamantly opposed to keep-it prototyping [A5, Tae91]. Boar's suggestions for rapid prototyping [Boa85] are often cited, and he generally recommends against keep-it prototyping. Guimaraes [Gui87] is more specific in suggesting that the prototype needs to be discarded only if it is used to test complex design alternatives.

Our sources do not support the notion that keep-it prototyping results in poor quality software products. In fact, ten of the studies specifically recommend keep-it (or *evolutionary*) prototyping. Only six authors insist that prototypes be discarded. Three of the case studies [AB90, Hek87, Tam82] represent successful keep-it prototyping on large software projects. On small projects, several sources suggest that keep-it prototyping is essential. Strand and Jones state that "for small-scale systems, clearly the prototype must be a part of the finished system or prototyping is economically infeasible" [SJ82]. Gupta et al. [GCG+89] report that "the environment should encourage code reusability, to extract maximum work from a minimum of code ... to avoid reprogramming as the system evolves." We conclude that both keep-it and throw-away prototyping have pitfalls. See Section 5 for details.

What language should be used to develop the prototype?

Although most sources stress the importance of carefully selecting a language suitable for prototyping, 22 cases employed 18 different languages. One common suggestion is that the language should offer convenient input/output development. Two cases [AB90, GCG+89], suggest object-oriented environments are preferred for evolutionary prototyping. The most popular single language choice was Lisp, although it was used in only three cases [HLC82, Hek87, AB90].

Can prototyping be used for developing large systems?

Of 24 cases, three can be considered large [AB90, Hek87, Tam82], and another eight are medium or medium-to-large [Gom83, BJK+89, CB85, A1, A4, Zel80, GCG+89]. We find no support for the common notion that *evolutionary* prototyping is dangerous for large projects. All three cases involving large projects used evolutionary prototyping. However, the pitfalls involved with evolutionary prototyping seem to grow in proportion to the size of the system being prototyped. See Section 5 for specific recommendations.

Does rapid prototyping require experienced programmers?

Eleven cases used experienced programmers and two used inexperienced programmers. Experience levels are not indicated in the remaining cases. One case utilized novice programmers successfully on a small system in a non-contractual environment [Ala84].

A few of the sources recommend an experienced, well-trained team as essential for successful prototyping. Connell and Brice [CB85] describe a project which failed partly because temporary student programmers were thrown into a rapid prototyping environment. Other sources [A7, Tam82, Ala84] state that experienced (or at least thoroughly trained) engineers are required for successful use of rapid prototyping. Alavi describes a successful small-scale project which utilized entry-level programmers, but then concludes without explanation that "Prerequisites to successful prototyping include ... motivated and knowledgeable users and designers" [Ala84]. Overall, the evidence suggests that it is dangerous to put inexperienced programmers into a rapid prototyping environment.

5 Pitfalls and How to Avoid Them

Much of the literature about rapid prototyping describes inherent pitfalls not found when using the specification approach [Boa85]. Examination of the case studies confirms that these pitfalls are real. Fortunately, the sources also describe similar methods of dealing with each of them.

Inferior design quality

Poor design quality commonly results when a prototype is meant to be thrown away, but is kept instead in order to save costs. Quality also suffers when, during evolutionary prototyping, design standards are not enforced in the prototype system. To avoid this problem, Connell and Brice suggest adhering to a design checklist [CB85]. Code which is transferred to the final product must satisfy the checklist. Quality can also be improved by limiting the scope of the prototype to a particular subset (often the user interface) [Ala84, Tam82, Tae91], and by including a design phase with each iteration of the prototype [Hek87]. Another option is to completely discard the prototype.

Unmaintainable code

A prototype which is developed quickly, massaged into the final product, and then hurriedly documented can be very difficult to maintain or enhance. The advice for avoiding inferior design quality also apply here. Documentation criteria should be included in the design checklist to ensure complete system documentation of the prototype. Other suggestions include frequent reviews [Hek87] and the use of object-oriented technology [GCG+89]. Of course, discarding the prototype is also an option.

Poor performance

The consensus is to build the prototype without initial concern for system performance [Gom83, Zel80, HLC82]. However, the prototype can demonstrate functionality that is not possible under real-time constraints. This problem may not be discovered until long after the prototype phase is complete. One way of avoiding this pitfall is to use an open system development environment to make it easier to integrate faster routines when necessary [GCG+89]. Two sources suggest the early measurement of performance, especially when evaluating design alternatives [CB85, Gui87]. Performance issues are less critical when the prototype focuses solely on the user interface. Again, discarding the prototype is also an option.

A throw-away prototype becomes the product

This common problem (observed by four sources [Gui87, CB85, A3, Tae91]) typically occurs when managers are initially sold on the idea of throw-away prototyping. But, when they see the prototype, managers decide to save money by massaging the prototype into the product. The resulting system often lacks robustness, is poorly designed, and is un-maintainable. [Gui87, CB85, A3, Tae91] stress the importance of avoiding this pitfall; *either plan to keep the prototype, or discard it.* One of the perils of throw-away prototyping is that the prototype may not get thrown away. Guimaraes observes this phenomenon "creating trouble at some companies" when "undocumented prototypes that were intended to be thrown away are kept, and become the poorly planned bases for large, complex systems that are consequently difficult to use and maintain" [Gui87]. Managers can avoid this pitfall by adequately training the programmers and maintaining a firm commitment to the prototyping paradigm. Careful definition of the scope and purpose of the prototype is also indicated as a means of avoiding this pitfall.

Lengthy iteration of the prototype phase

Prototype development can be time consuming, especially when the purpose and scope of the prototype is not initially well-defined. Boar's work [Boa85] describes how inadequate narrowing of the scope of the prototype can lead to thrashing or aimless wandering, the result of which will be of little use to a design team later on. Four of the industry sources support Boar's claim [Ala84, CB85, HLC82, Tam82]. Additional suggestions for avoiding this pitfall include using a highly disciplined approach to scheduling prototyping activities such as described in [Hek87], and avoiding throwing entry-level programmers into a rapid prototyping environment [CB85].

Skeptical end-users

End-users can become skeptical when given too much access to the prototype. Users may equate the incompleteness and imperfections, which naturally exist in a prototype, with shoddy design.

By limiting user interaction to a more controlled setting, user expectations can be kept at reasonable levels [CB85]. Two sources [Ala84, A3] recommend not overselling the prototype.

Evolving a large system results in a large mess

Evolutionary prototyping on large projects can result in a system filled with patches as hastily-designed modules become the root of later problems. One way to avoid this is to use an object oriented approach [AB90, GCG+89, A5]. Another method is to limit prototyping to user interface modules which are less likely to involve important data structure design decisions. A highly disciplined approach such as that used in [Hek87] is also recommended.

Underestimating conversion time

Prototyping languages are often utilized to ease implementation of a particular aspect of the system. For example, if the prototype is developed to test various user interface options, a language which provides convenient I/O capabilities is selected. The conversion may be non-trivial if the ultimate target language does not have such simple I/O handling. This pitfall was observed by Zelkowitz [Zel80], and alluded to by Taenzer [Tae91]. Careful definition of the scope of the prototype, and a systematic comparison of the features of both languages can help to avoid this pitfall.

6 Conclusions

The real-world case studies suggest that rapid prototyping, when employed properly, leads to improved software quality. The primary improvements are ease of use, better match with user needs, tighter code, and often better maintainability. We do not find support for the common notion that rapid prototyping cannot be used for developing large systems. We find no particular bias towards either keep-it or throw-away prototyping, and the case studies provide little insight into which languages are better for prototyping. We find a number of inherent pitfalls with prototyping. In order to avoid these pitfalls (described in Section 5), we recommend that developers try the following:

- Carefully define the purpose and scope of the prototype.
- Avoid the use of entry-level programmers.
- Utilize a design checklist.
- Use an open system environment (such as object-oriented methods).
- Consider performance issues early.
- Limit end-user interaction to a controlled setting.
- Do not under-estimate conversion time.
- *Do not* keep a prototype that was not initially intended to be kept.

Case study data is not easy to find and is somewhat biased. Negative results are seldom published. Our analysis can be improved with additional case study data, especially descriptions of rapid prototyping in large software projects. Although we cannot conclude that rapid prototyping

is dangerous for large projects (the data indicates otherwise), we are not willing to state that rapid prototyping is good for large projects using information from only three case studies.

Rapid prototyping is being successfully employed in the software industry. With the lessons provided by the case studies, rapid prototyping can be used to improve software quality.

References

[Boa85] B. Boar. *Application Prototyping - A Project Management Perspective.* AMA Membership Publications Division, 1985.

[Bro75] F. Brooks. *The Mythical Man-Month.* Addison-Wesley, 1975.

[Har87] K. Harwood. On prototyping and the role of the software engineer. *ACM SIGSOFT Software Engineering Notes*, 12(4):34, Oct 1987.

[LB89] J. Leszczykowski and J. Bieman. Prosper: A language for specification by prototyping. *Computer Languages*, 14(3):165–180, 1989.

[Mit82] R. Mittermeir. Hibol – a language for fast prototyping in data processing environments. *ACM SIGSOFT Software Engineering Notes*, 7(5):133–141, Dec 1982.

[Pat83] B. Patton. Prototyping – a nomenclature problem. *ACM SIGSOFT Software Engineering Notes*, 8(2):14–16, April 1983.

[Rat88] B. Ratcliff. Early and not-so-early prototyping – rationale and tool support. *Proc. COMPSAC 88*, pp 127–134, Nov 1988.

Case Studies

[AB90] E. Arnold and D. Brown. Object oriented software technologies applied to switching system architectures and software development processes. *Proc. 13th Annual Switching Symposium*, vol 2, pp 97–106, 1990.

Describes a methodology for applying object oriented technology to switching systems. The advantages of an object oriented approach as it relates to prototyping is discussed.

[Ala84] M. Alavi. An assessment of the prototyping approach to information systems development. *Communications of the ACM*, 27(6):556–563, June 1984.

Twelve information systems development projects using the prototyping approach in six organizations are analyzed. Also, an experiment involving nine student groups is presented which compares the use of the prototyping method with the specification approach.

[BGS84] B. Boehm, T. Gray, and T. Seewaldt. Prototyping versus specifying: A multiproject experiment. *IEEE Transactions on Software Engineering*, SE-10(3):290–302, May 1984.

Seven student teams developed the same product, three use prototyping, and four use specification. Boehm compares the specification technique with the prototyping technique.

[BJK+89] P. Bonasso, P. Jordan, K. Keller, R. Nugent R. Tucker, and D. Vogel. A software storming approach to rapid prototyping. *Proc. 22nd Annual Hawaii Conf on System Sciences*, vol 2, pp 368–376, 1989.

A new methodology for rapid prototyping called *software storming* is proposed, which involves an intense one-week period of videotaped interaction between software designers and end users.

[CB85] J. Connell and L. Brice. The impact of implementing a rapid prototype on system maintenance. *AFIPS Conference Proceedings*, vol 54, pp 515–524, 1985.

Describes case studies using the INGRES data base system. One project was completed and implemented to the apparent satisfaction of all parties. Another was criticized by various parties and eventually abandoned.

[FM82] R. Ford and C. Marlin. Implementation prototypes in the development of programming language features. *ACM SIGSOFT Software Engineering Notes*, 7(5):61–66, Dec 1982.

Describes the authors experiences applying prototyping techniques to the development of programming languages with advanced features.

[GCG+89] R. Gupta, W. Cheng, R. Gupta, I. Hardonag, and M. Breuer. An object-oriented VLSI CAD framework. A case study in rapid prototyping. *IEEE Computer*, 22(5):28–36, May 1989.

Shows the suitability of rapid prototyping for the development of CAD systems, and to object oriented development. The focus is on the specific application being prototyped, rather than the merits of prototyping in general.

[GHPS79] G. Groner, M. Hopwood, N. Palley, and W. Sibley. Requirements analysis in clinical research information processing – a case study. *IEEE Computer*, 12(9):100–108, Sept 1979.

A number of small clinical research programs were developed and evaluated for users who have difficulty in clearly expressing their computing needs.

[Gom83] H. Gomaa. The impact of rapid prototyping on specifying user requirements. *ACM Software Engineering Notes*, 8(2):17–28, April 1983.

Describes a case study of a large system developed at General Electric called PROMIS, a Process Management and Information System for integrated circuit fabrication. The progress of prototyping and subsequent development are tracked by the author.

[GS81] H. Gomaa and D. Scott. Prototyping as a tool in the specification of user requirements. *5th Int. Conf on Software Engineering*, pp 333–339, 1981. See [Gom83] for description.

[Gui87] T. Guimaraes. Prototyping: Orchestrating for success. *Datamation*, pp 101–106, Dec 1987.

Describes an extensive field study of 48 Fortune 1000 companies which evaluates whether or not prototypes should be discarded. Although the majority of companies employ throw-away prototyping, the author concludes that generally keep-it prototyping is preferable.

[Hek87] S. Hekmatpour. Experience with evolutionary prototyping in a large software project. *ACM SIGSOFT Software Engineering Notes*, 12(1):38–41, Jan 1987.

Rapid prototyping methodologies are used to develop a prototyping language called EPROL and a prototyping system called EPROS. Shows the successful use of evolutionary prototyping on large systems.

[HLC82] C. Heitmeyer, C. Landwehr, and M. Cornwell. The use of quick prototypes in the secure military message systems project. *ACM SIGSOFT Software Engineering Notes*, 7(5):85–87, Dec 1982.

A small military prototype messaging system is developed using LISP. Focuses on the role of the prototype, prototyping efforts in general, and reuse of prototype code.

[Rze89] W. Rzepka. A requirements engineering testbed: Concept, status and first results. *Proc. 22nd Annual Hawaii Conference on System Sciences*, vol 2, pp 339–347, 1989.

Describes a prototyping environment at the Rome Air Development Center. Concentrates on a heavily I/O intensive application, and notes significant speedup in development time.

[SJ82] E. Strand and W. Jones. Prototyping and small software projects. *ACM SIGSOFT Software Engineering Notes*, 7(5):169–170, Dec 1982.

The retention of the prototype code and the use of a special-purpose prototyping language are shown to be useful techniques in small-scale software projects.

[Tae91] D. Taenzer (U.S. West). Personal communications.

Rapid prototyping improved quality and ease of use of final products, and increased user participation. Developers are often pressured into reusing a throw-away prototype. Careful definition of the scope and definition of the prototype is recommended.

[Tam82] D. Tamanaha. An integrated rapid prototyping methodology for command and control systems: Experience and insight. *ACM SIGSOFT Software Engineering Notes*, 7(5):387–396, Dec 1982.

Describes a major ($100M) military project implemented successfully using rapid prototyping. CICS is used as the development environment. Provides insights into the effects that prototyping necessitates on management techniques and the software development process.

[Zel80] M. Zelkowitz. A case study in rapid prototyping. *Software – Practice and Experience*, 10(12):1037–1042, Dec 1980.

Describes experiences implementing Backus' FFP System using rapid prototyping. The need for certain functionality became apparent while exercising the prototype.

Sources Requesting Anonymity

[A1] Employee at major university.

Development of a University online registration system is described. Authors report improved customer satisfaction, ease of use, and ease of training.

[A2] Researcher at major university.

Expressed satisfaction with SCHEME as a prototyping language, based on the ultimate production of a working system in C.

[A3] Engineer at large telecommunications firm.

Describes problems with rapid prototyping which initially stemmed from management not understanding the limits of a prototype, and which have caused hardships and ultimate failure.

[A4] Engineer at large military contracting firm.

A substantial improvement in product quality, reduced effort, lower maintenance costs, and faster delivery is achieved by the use of rapid prototyping. In particular, leveraging with off-the-shelf products helps greatly.

[A5] Engineer at large data processing firm.

Prototyping has been quite effective. Recommendations include using object oriented approach, throw-away prototyping, and the careful selection of an appropriate prototyping language.

[A6] Engineer at large Government/Military division.

Small government systems have been developed successfully with rapid prototyping. Reuse of 50% of the prototype was generally achieved. Product quality was improved, and users were more likely to get what they wanted. Some people became upset when their ideas were quickly discredited by experiences with the prototype.

[A7] Engineer at small software company.

Prototyping worked well in a small project environment. Lines-of-code per day can be bid at a higher rate, but the method only works if experienced engineers are available.

Michael Perdue
Sun Technology Enterprises, Inc.
2550 Garcia Ave.
Mountain View, CA 94043

Bio

Michael Perdue is Manager of Product Verification for the Software Development Products business unit in Sun Technology Enterprises, Inc. He has been working on implementing a new development process for Sun Microsystems over the last three years. He is especially interested in the human side of processes and how people adapt to change. He was previously involved in software quality improvement programs at Motorola

Sun has been working 3-1/2 years to effect a culture change in the development of software products. This talk will describe our experiences (good and bad) and key lessons we've learned. It will focus on how management teams and the various engineering communities within Sun are reacting to the new process.

NOTE: Paper unavailable by printing deadline.

CMSYS - A Database System for Software Metrics Collection

Geoff Flamank
Software Quality Assurance
Dynapro Systems Incorporated
800 Carleton Court
Annacis Island, New Westminster
British Columbia, Canada V3M 6L3
(604)521-3962

Abstract

Collecting the desired data is a major problem when attempting to analyze information (metrics) and improve a software development process. Many organizations use a paper-based system which does not promote efficient retrieval of information and often results in issues and suggested changes being lost or forgotten. A problem with many of these systems is that they divorce themselves from the software development process. ***CMSYS*** (Change Request and Configuration Management System) - a change control and metrics repository - was designed and built to have this data collected in real-time by being an integral function of the regular engineering change and configuration management cycle.

This paper is a case study which discusses the ***CMSYS*** database design focusing on key concepts such as: Customer Impact Severities and Weights, Failure and Fault Weights, Identification of Discovered and Responsible Processes, and Review and Suggestion tracking. Two valuable software metrics that can be easily produced using ***CMSYS*** are described, along with examples of their use : the *Quality Index* is used to monitor **product** quality throughout the development life cycle; the *Review Efficiency Index* is used to monitor the development **process** and its effectiveness.

Biographical Sketch :

Geoff Flamank is a Software Quality Assurance Engineer at Dynapro Systems Incorporated (DSI) based in Vancouver, British Columbia, Canada. With a development staff of over 50 people, DSI designs and manufactures hardware, software and embedded firmware for real-time industrial process control applications. Previous to his work in SQA at DSI, Mr. Flamank worked as an SQA Engineer in the building of the Automated Weather Distribution System for the United States Airforce. He also held positions as a software engineer and a project manager. His work in the development and implementation of ***CMSYS*** earned him the 1991 Dynapro Award for Quality. Mr. Flamank's interests are in software metrics, the software development process, and Configuration Management.

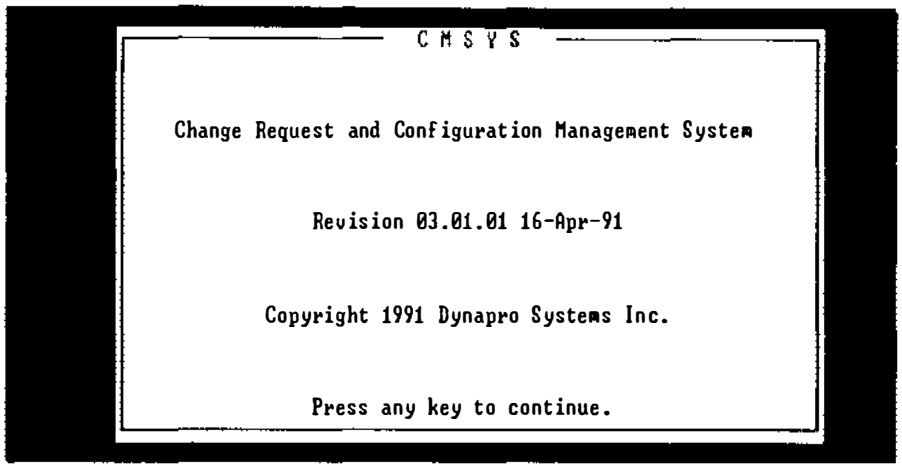


Figure 1 - CMSYS

The topic of software metrics is often discussed at many conferences. One fact boldly stands out as its major obstacle: collecting the data is the most difficult part of metrics analysis. ***CMSYS*** (Change Request and Configuration Management System), a sophisticated automated change control system developed in-house at Dynapro Systems Incorporated (DSI), has allowed us to collect software process metrics effectively and efficiently by integrating a disciplined Configuration Management process with a comprehensive, multiuser Change Request database design and a delta-storage mechanism. It differs from other systems in that the stored change information facilitates defect analysis, error trend analysis, software process and product metrics analysis, quality goal setting and maintenance planning - in addition to the more traditional source code and document version control.

The Dynapro software quality improvement process is comprised of five major steps:

(1) define the processes and deliverables,
(2) establish a strong Configuration Management (CM) organization,
(3) control and monitor the changes,
(4) set some goals, and
(5) evaluate the goals continuously and make adjustments where necessary.

CMSYS plays a major role in the achievement of these steps and is considered the cornerstone of our software quality improvement program.

This paper will present some background on the DSI development process, a terse description of the overall system architecture and table structures, followed by an in depth discussion of how, when and what data is collected, some important design considerations and what we do with the information. The successes, challenges and future plans will also be discussed.

A. Background

The pre-1987 phase at DSI consisted of a non-networked environment of PC's in which one of our major products still operated in the CP/M environment! The manner in which a product was built and the overall quality of the products was left up to a small group of highly skilled, knowledgeable individuals. Configuration Management (CM) was

performed in an adhoc manner by backing up diskettes and sharing the information by swapping diskettes - the classic, exciting "small company" where everyone knew what was happening but could not quantify it. One of the difficulties in quantifying quality concerned the handling of changes. The monitoring and logging of problems was handled via a paper-based system which required numerous approvals by people outside of the development cycle (Senior Managers and Project Managers) before change could happen. The photocopied form was passed by hand to team members (often three to four sets of hands before it reached the actual implementor of the change or fix). Non-current Summary Reports were generated weekly then discussed in a multi-project meeting. Needless to say, this was not an efficient or highly reliable system. Nor did it give us the flexibility and means to ask the questions and get the answers to what we needed to know. Namely, "how do we improve the way we are building our software products?".

Until the mid-80's, DSI's product quality was measured by:

(1) the number of sales,
(2) the number of stop-shipments,
(3) the number of requested product enhancements, and
(4) the number of requests for new product development.

These, however, are not enough to survive in today's competitive market place. Customer expectations for functionality, reliability, usability, correctness, overall polish, and speed of repair response are all increasing. We must not only build a higher quality product, we must get it to the customer quicker.

The theme that we have put in place at DSI is:

"If the process is done correctly with the highest level of commitment, the result will be a higher quality product which gets to the customer quicker".

This has been a strong challenge to our development team and process.

In 1987 a new project began which introduced the concept of customer quality requirements - building a product to meet defined standards. This marked the first occurrence of a disciplined software process integrated with a Configuration Management organization. Though development took place in the PC environment, code was stored and version-controlled on a micro-VAX using the DEC Code Management System (CMS) and changes were controlled via a VAX-resident system developed in-house. These changes were successful in that they introduced DSI to CM. The change control logging process, however, was not entirely accepted by the developers because it:

(1) did not reside on the development environment,
(2) was relatively slow compared to existing tools used, and
(3) had a poor user-interface.

The search then began for an integrated Configuration Management and Change Request System that would satisfy what we thought were some fairly basic needs - the system had to:

(1) be a multi-user, on-line system,
(2) allow for tailoring to our development process,
(3) be easy to use,
(4) be flexible to allow for user-definable information retrieval, and
(5) be resident on a Novell Development Environment platform.

After much investigation, we realized that we weren't going to find what we wanted and began building a system which integrated a delta-storage mechanism with a multi-user, relational database.

The 1991 DSI software development team is relatively small - approximately 50. Its size, corporate structure and people are all leading factors in the success of ***CMSYS***. ***CMSYS*** is used, on the average, 30 to 40 times per day.

B. System Architecture

Figure 2 shows a Context Diagram for ***CMSYS*** indicating the various departments and team members actively using the system, and the external software packages tied to the system.

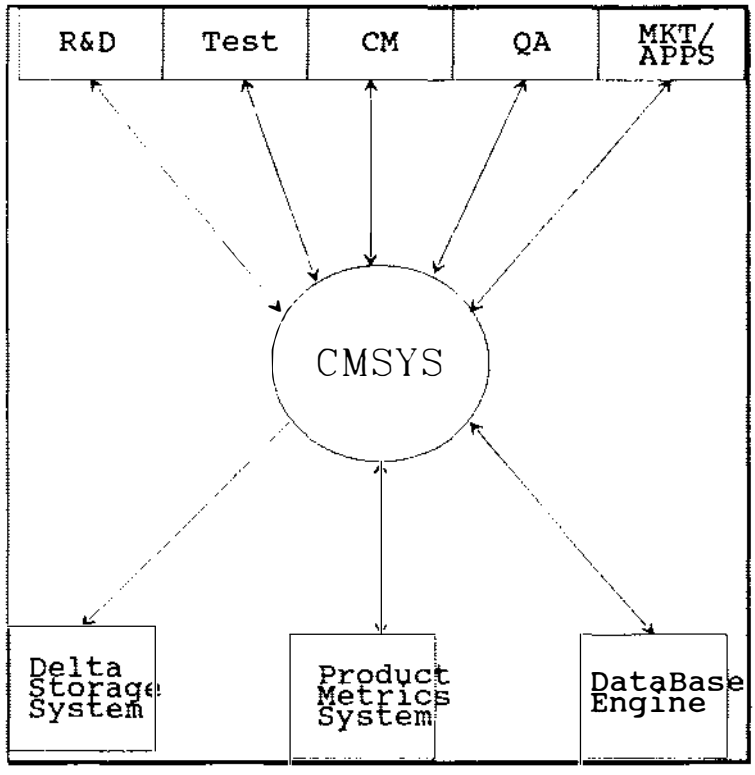


Figure 2 - Context Diagram

CMSYS provides the ability to monitor the quality of software processes by viewing different aspects of the change activity. Engineers assigned to develop and maintain the software for a specific subsystem or unit may select change activity for that specific item and act accordingly. Project Leaders or System Architects may wish to view activity at the subsystem level. Quality Assurance and Engineering Methodologists may wish to view the project quality at a process level. Senior Management may only wish to view the project at a "Number of Problems" level.

CMSYS interfaces with a delta-storage system which handles version control and storage of all code and document changes required to generate a specific release. All unit code is "reserved" and "baselined" by the Configuration Management organization through this *CMSYS* interface.

CMSYS generates **process** metrics and, in the *CMSYS* database environment, provides the opportunity to research and correlate imported externally generated **product** metrics. The product metrics are generated as selected Releases are built and baselined. An external package, **PC-Metric** from SET Labs, is used to generate software product metrics. The metrics generated include Halstead's Software Science, McCabe's Metric, Lines of Code, and others. The results are imported into the *CMSYS* database environment where further off-line metrics analysis can be performed.

To promote flexibility and ease of maintenance, a leading DOS-based, multi-user, relational database package was chosen to quarterback the system. Through interaction with external code control packages and product metric collection tools, real-time information on the change activity and quality of a product during development and maintenance is made available to Software Engineering, Quality Assurance/Test Engineering, Configuration Management, Marketing, or any other team member.

Once the development process becomes disciplined, all that remains for successful metrics generation is to ensure that the right information is collected.

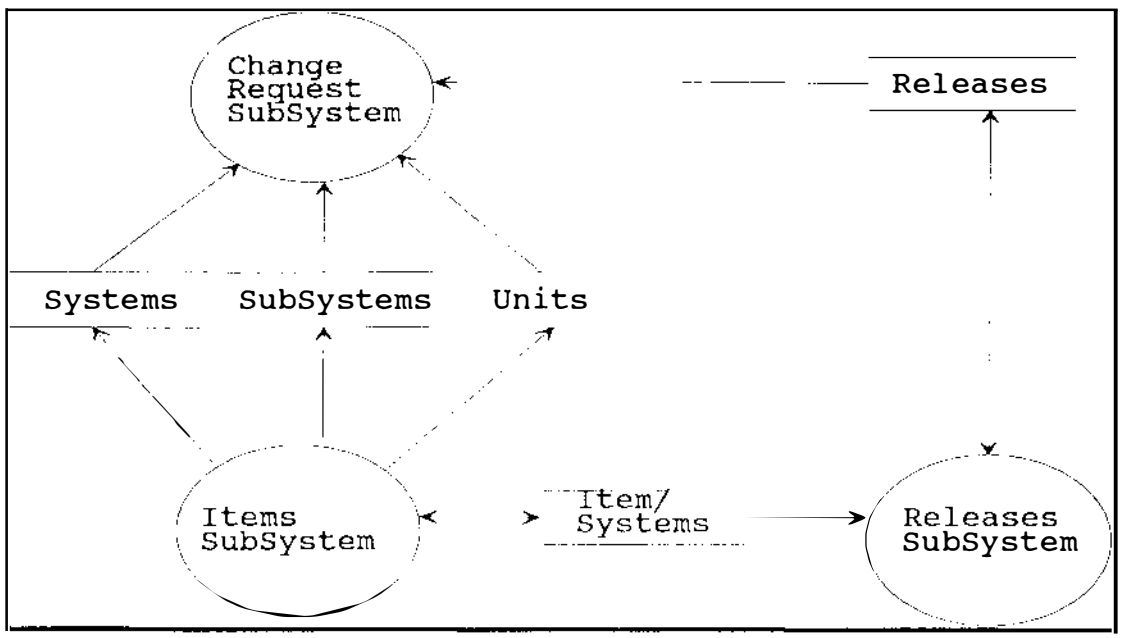


Figure 3 - CMSYS Subsystem Architecture

Figure 3 displays the major *CMSYS* subsystems and their data interfaces. More information on the Items and Releases subsystem can be obtained by contacting the author.

C. The Change Request

CMSYS and the Change Cycle

It is absolutely fundamental to the success of a software metrics collection system that all of the change information be entered without redundancy. That is, all change activity should be initiated through the entry of a new Change Request (CR) with subsequent change cycle information appended to this record and only this record. This implies that there is one single source of information that is maintained and made available for information retrieval.

Equally as important is that the system be accepted by the software engineering team. In order for this to happen, the information must be valuable, current, and accessible at any time. Figure 4 shows the Change Cycle at a very simplified level and the *CMSYS* involvement.

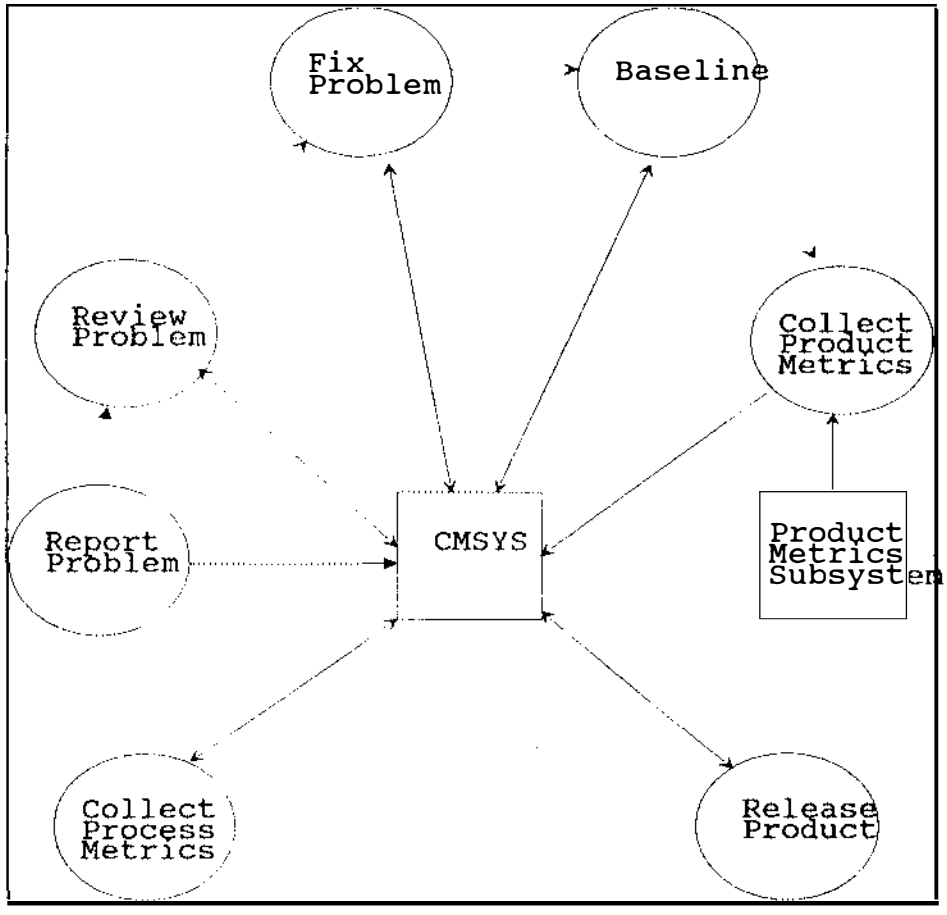


Figure 4 - Change Cycle

Change Request Entry

Figure 5 displays the standard entry form for a Change Request (CR) - it is used for virtually all CR activity. This form is comprised of three sections:

(1) general information (title, type, severity, etc. - the top third of the screen),
(2) change description (middle third), and
(3) dependancies (items that are affected by the change).

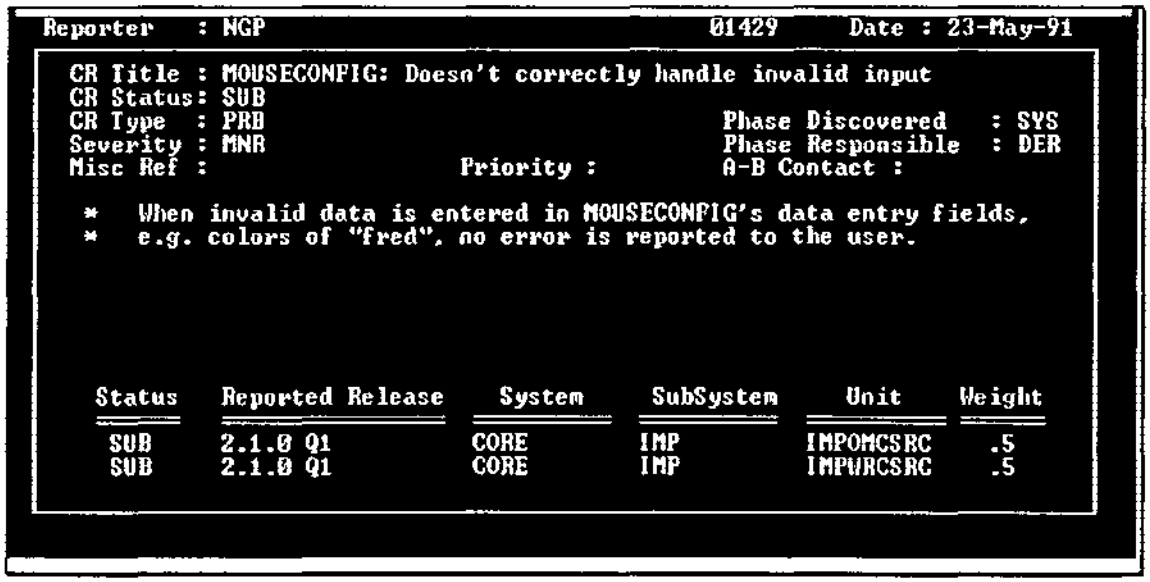


Figure 5 - Change Request Entry

A brief description of the key fields follows:

Product (not shown)

Each product, logically and physically, has its own metrics change information tables. Software engineers do not have a great deal of interest in the status of projects other than their own, nor do they wish to have the performance of their activities affected due to the processing of other project data.

CR Title

A one line description is used for quick searches and reports.

CR Type

CMSYS handles the activities associated with most types of changes. We are currently using it to track suggestions, planned updates (changes to an established baseline as a result of enhancements), and review minutes, as well as problems.

CR Severities

Each Change Request can be assigned a severity. Though problems are, generally, the only entities assigned a severity, ***CMSYS*** allows the user to apply it to other types of changes. Each severity is given a weight which is later used to calculate the Quality Index (see the section on Quality Metric Graphs). The following Severities and Weights are used:

- Critical 10
- Major 3
- Minor 1
- Incidental .5

The weightings are loosely based on the IEEE Standard Dictionary of Measures to Produce Reliable Software, Std. 982.1-1988, 4.8 Defect Indices.

Phase Discovered/Phase Responsible

A key feature of ***CMSYS*** is its ability to identify the process or phase in which a problem was discovered compared against the process or phase when it should have been discovered. If each phase is assigned a process value, the difference between phase discovered and phase responsible will indicate the number of "checks" that failed to identify this problem. When these differences are summed and the average calculated, an interesting number we call the "Review Efficiency Index", is generated. This new measurement permits us to monitor the efficiency of the development process. Further discussion on this concept can be found in the section on Quality Metric Graphs.

The development processes, problem types and process values are :

Phase	Types	Values
Requirements	User Interface Missing Requirement Misinterpreted/Incomplete Miscellaneous	1
Design	Missing Design Feature Interface Interprocess Handling Logic/Processing Data Structures Error Handling Design Architecture Miscellaneous	2
Implementation	Data Structures Logic(If-Then-Else, etc.) Computation Error Handling Miscellaneous	3
Product Builds	Missing files, wrong environment	4

Misc. Ref. Priority, A-B Contact

These are miscellaneous data fields which are specific to our company's process and business relationships.

CR Descriptions and Solutions

CR Description text and Solution text (not shown) are stored in separate tables. This allows users of the system to establish their own criteria for key-word searching.

Failure and Fault Tracking - The "Dependancies"

The Dependancy Table entries are found in the bottom third of the screen in Figure 5. The fields that are displayed on this form are:

Statuses

The overall status of a CR is determined by the lowest status of all items in the Dependancy Table. ***CMSYS*** uses the following statuses and identifies each with a sequence or weight:

- Reported	0
- Reviewed	1
- Assigned	2
- Reserved	3
- Submitted	4
- Delta-stored	5
- Deferred	6
- Baselined	7
- Released	8
- No Action Taken	9
- Withdrawn	10
- Transferred	11

Reported Release

A "Reported Release" indicates the release in which the change was identified. Release identifiers at Dynapro have the following convention:

- 1.0.0 E1...n	Used for Engineering Integration Builds
- 1.0.0 Q1...n	Used for System Testing and Acceptance Testing Builds
- 1.0.0	Release delivered to the Customer

The "1.0.0" portion of the release identifier is substituted with the specific release that we are working towards. That is, if we are currently working towards enhancing an existing product and the release of this product will be identified as "3.1.2", all Integration or Engineering releases will be of the format "3.1.2 E1...En", all formal System Testing/Acceptance Testing builds will be of the format "3.1.2 Q1...Qn".

System, Subsystem and Unit

The components that are controlled by ***CMSYS*** are identified in the Item/Systems Table. This table allows for a 3-level hierarchy to define the software: Systems, Subsystems or Units. A "System" is defined as a deliverable and is always the item to which a formal "Reported Release" identifier is assigned. The "Subsystem" is used to partition subordinate "Units" into logical groups and is used generally for reporting purposes only. The "Unit" is the main working piece in ***CMSYS***. It consists of one or more files which form a logical relationship.

The minimum data entry requirements for a dependancy item are the "Reported Release" and the "System". When a change is completed and submitted to Configuration Management, the associated units are added to the Dependancy Table for the specified CRs.

Dependancy Weights

A "Weight" is established, by the project engineer, for each dependancy item ("fault") associated with the CR. This weight can be distributed evenly or selectively valued for each item to identify its true impact.

The sum of the weights of the faults should always equal one for each **Development Line.** A Development Line consists of all releases associated with a current or variant line of development. For example, if internal releases 1.0.0 E1, 1.0.0 E2, and 1.0.0 Q1 were required in order to ship an external version called 1.0.0, then those releases are deemed to be part of the 1.0.0 Development Line.

Calculations of CR quantities are generated by using the **weight** rather than the **number** of CRs in the Change Request Master Table. This allows for identification and tracking of partial change implementation for variant development lines.

Status Changes

The Change Request screen (Figure 5) is also used by the Project Engineer to update the CR for the Review, Assign, Defer, Withdraw, Transfer and No Action Taken statuses. Configuration Management uses this screen for updating Baseline and Release statuses.

Miscellaneous Data

Other Dependancy data not shown here are:

- Reviewer, Review Date
- Deferrer, Deferred Date, Deferred-To Date
- Withdrawer, Withdrawn Date, Comment
- Assigner, Assigned Date, Assignee, Fix For Date, Fix For Release
- Fixer, Fixed Date, Fix Man Days
- Fix Man Days
- Baselined Release Identifier
- Customer Release Identifier

Figure 6 shows all of the fields used for each Change Request and dependant item.

D: Retrieving Specific Change Request Information

Querying the System

CMSYS provides a query mechanism which allows the user to retrieve almost any combination of information from the database.

For example, the Selective Query shown in Figure 6, would retrieve all of the "MAJOR (MAJ) " "PROBLEMS (PRB)" reported during the "SYSTEM-TEST (SYS)" phase which were associated with the "CORE" System and released to the customer in release "2.1.0". This retrieval mechanism is very flexible. *CMSYS* provides pre-defined queries such as "Active" (all Change Requests not yet Baselined) and "Full" (all Change Requests regardless of their status).

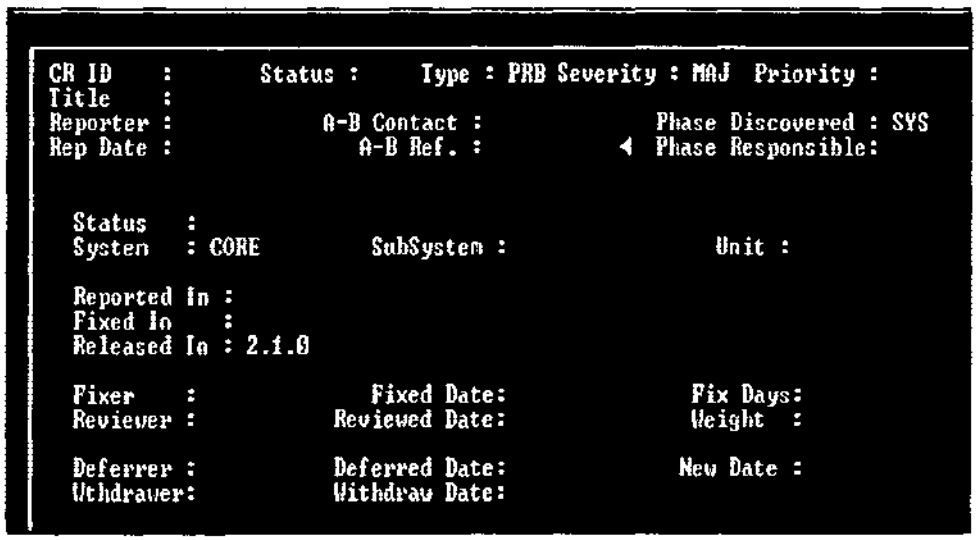


Figure 6 - Selective Query

Feeding Back Into the Process

The primary function of *CMSYS* is to collect useful information about the processes and then feed it back in a form which promotes their improvement. An effective method of doing this is to run a causal analysis of a specific system, subsystem or unit and determine the processes that have failed, the types of defects injected during the process and, more importantly, how badly they have failed. Figure 7 - Causal Analysis - shows the output of a specific area. This output can then be used, for example, to generate initial review checklist criteria or to determine error tendancies for specific modules and/or developers.

Description	CRT	MAJ	MNR	INC
Design - Interface		11	0	0
Design - Design Architecture		3	2	0
Implementation - Logic (If-Then-Else problems, decisions...)		2	0	0
Implementation - Miscellaneous		1	4	2
Implementation - Data Structures (Runaway pointers...)		1	3	0

Figure 7 - Causal Analysis

Quality Metric Graphs

An exciting feature of *CMSYS* is its on-line graphing capabilities. Using this, four graphs depicting different views of the current quality are provided:

(1) Change Request Summary (Figure 8),
(2) Problem Distribution (Figure 9),
(3) Review Efficiency Index (Figure 10), and
(4) Quality Index (Figure 11).

Change Request Summary

The **Change Request Summary** (Figure 8) graph provides the project engineer with a visual feel for the status of all change requests in the system and to determine the amount of change management that is required. This is generally performed on a daily basis by the Project Engineer, Project Leader or the engineer responsible for administrating the change request database.

Problem Distribution

The **Problem Distribution** (Figure 9) graph indicates where active defects are located within the systems, subsystems or units. It is helpful in estimating maintenance and testing effort as well as pointing out quality problem areas. This graph is used by Quality Assurance to determine the relative quality levels of a system and by Test Engineering to determine any possible areas for future testing. Although this graph is usually generated immediately following a release, it can be used on an ongoing basis to determine defect distributions.

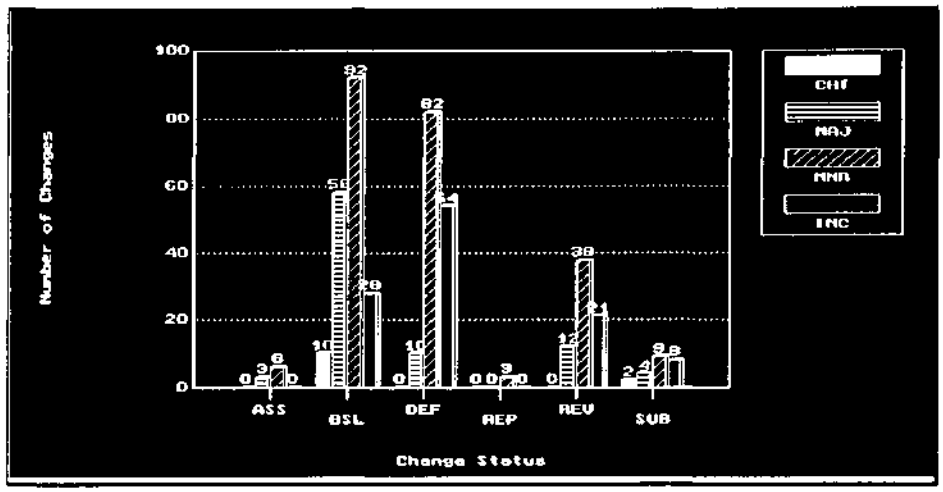


Figure 8 - Change Request Summary Graph

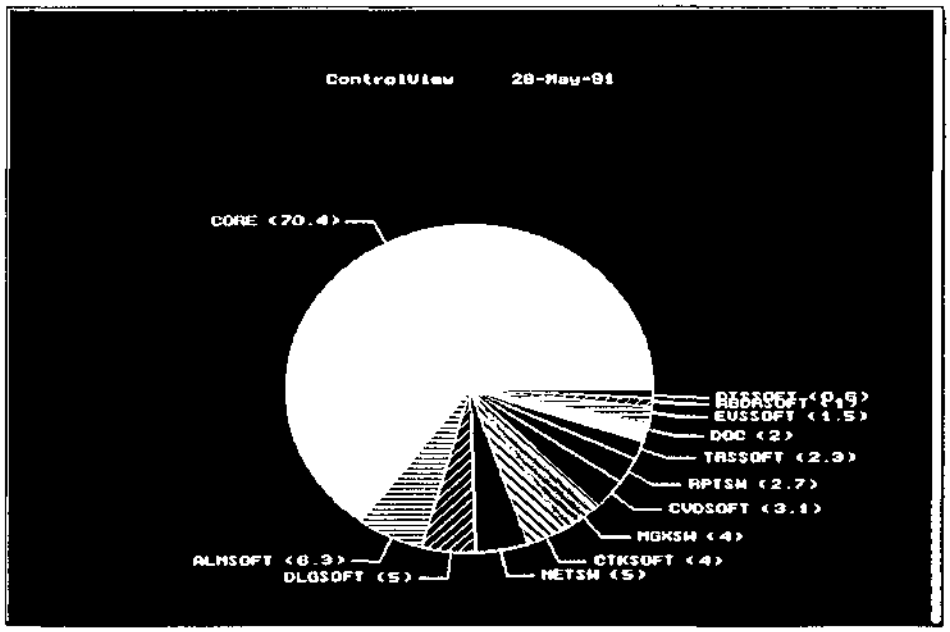


Figure 9 - Problem Distribution Graph

Review Efficiency Index Graph

CMSYS generates a **Review Efficiency Index** (Figure 10) indicating the effectiveness of the software review process. It is calculated by subtracting the originating phase value from the discovered phase value, summing this for all problems, then dividing the result by the total number of problems. For example, if a Requirements (valued "1") problem is found in the Implementation process (valued "3"), their difference would be "2". This essentially means that two review processes failed to detect the problem. If this were the case for all existing problems in the database, the **Review Efficiency Index** would indicate

"2". The ideal number would be "0" indicating that all problems generated during a specific process were found in that same processes review.

The series labelled "Average" is a running average calculated from a given start date while the series labelled "Actual" is the actual value for the monthly range.

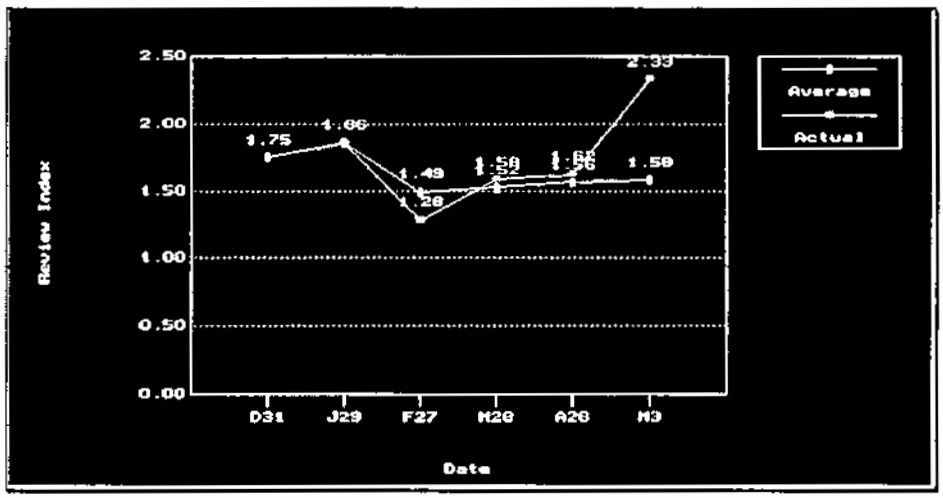


Figure 10 - Review Efficiency Index Graph

Quality Index Graph

CMSYS derives a **Quality Index (QI)** (Figure 11) by assigning a weight to a problem based on its criticality to the user. The graph shows three items: "Reported", "NotFix", and "NotFix QI".

The "**Reported**" value indicates the number of problems reported during the past week or month range.

The "**NotFix**" value indicates the number of problems which have not been integrated into a specific internal release or baseline. Problems which are in the process of being fixed and will tentatively be integrated into an upcoming release are still counted as being "Not Fixed".

The "**Not Fix QI**" value counts those problems "NotFix" by counting their severity weights. For example, if a problem exists with a severity of critical (Weighted "10"), this would be reported as "1" in the "NotFix" value but reported as "10" in the "NotFix QI" value.

This graph plots data on a weekly, monthly or quarterly detail level and, as is the case for all *CMSYS* functionality, is always current and on-line. When coupled with Lines of Code (or the metric of your choice), quality levels can be monitored and baselined then used for comparative analysis of projects, systems, subsystems, or units. The results can be used for establishing future quality goals. We call this a *"QI Density"*.

This graph is generated by the Project Engineer, Testing Engineer and Quality Assurance both prior to and directly following a release to determine the current quality level of a system, subsystem or unit.

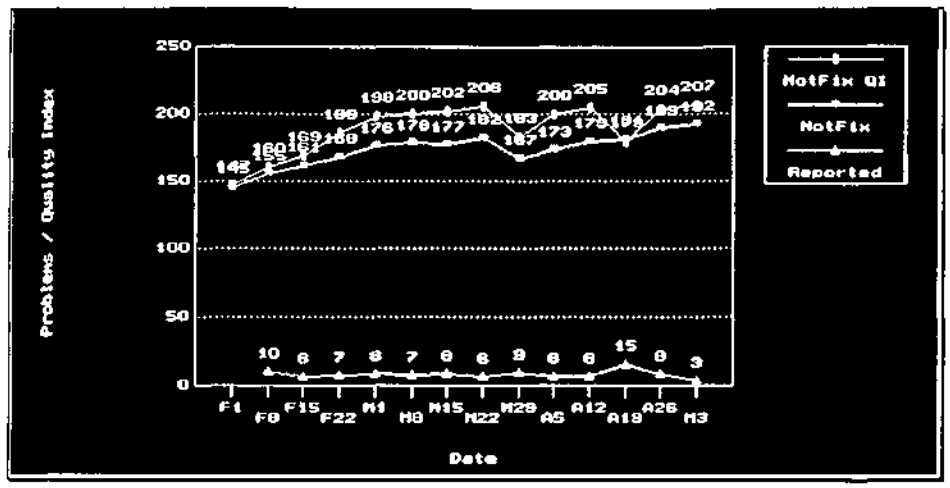


Figure 11 - Quality Index Graph

E. Successes and Challenges

Successes

We are currently using the **Quality Index** to monitor the quality levels of specific products in the field and under development. New project plans now include a target Quality Index. As a company, we have set a target of 50% improvement in delivered QI Density for each future release of these products. This sets an important precedent for our company and establishes a quantifiable quality goal to which we can strive.

DSI is using the **Review Efficiency Index** and has raised the goal for 1991 development to "1.0" from "1.75". This means that we are striving to catch all problems in the next phase's set of reviews. ***CMSYS*** provides the capability to monitor our development processes very closely to ensure that they remain under control.

A major success of ***CMSYS*** has been the shifting of problem handling from paper shuffling at a departmental level to resolution at the software engineering level. All team members have the ability to view the level of quality of their system, subsystem or unit at any time they desire and to respond efficiently. Senior managers can now get the "big picture" rather than being overwhelmed with "snap shots". Having a paperless system has completely eliminated any lost or misplaced Change Requests and has tightened up our maintenance and enhancement cycle. We can now plan our future maintenance and enhancement efforts effectively.

The generation of Version Description Documents (identifying changes between releases of a product) is now a simple, quick process. In the past, this function had been a painful one - it now can be performed in minutes as opposed to days.

With ***CMSYS***, we are charting and prominently displaying the following metrics on a monthly basis:

- Quality Index Density
- Review Efficiency Index
- Percent of Suggestions Implemented

Challenges

Our next step will be to organize our testing information into a more efficient, reusable form and link it to the ***CMSYS*** environment.

On-line correlation with product metrics (Halstead's Software Science, McCabe's Metric, LOC, ELOC, etc) still needs further investigation. We are still new at collecting this data and expect that in a year or so we will know whether or not any correlations are strong enough to be of use.

The question of how Object Oriented Programming will effect Configuration Management needs to be answered. How will you report changes in a SmallTalk or C++ environment? Increased reusability requirements might result in more changes to a given unit of code to make it worthy of reuse. Is this good or bad or merely a characteristic that requires tracking in some fashion? What is a unit in an Object Oriented environment? We look forward to the challenges.

Summary

This paper and its tangible results has demonstrated that collecting software process and product metrics can be successfully automated and integrated into the day to day development process. These metrics can then be used to drive improvements to the entire software development process.

HIGH INTEGRITY SOFTWARE STANDARDS ACTIVITIES AT NIST

John C. Cherniavsky, National Science Foundation
Washington, DC 20550

D. Richard Kuhn
Dolores R. Wallace
National Institute of Standards and Technology
Gaithersburg, MD 20899

ABSTRACT

This paper provides information about the National Institute of Standards and Technology (NIST) effort to produce a comprehensive set of standards and guidelines for the assurance of high integrity software. In particular, the paper presents the results of a Workshop on the Assurance of High Integrity Software held at NIST on January 22-23, 1991 and activities at NIST in support of assuring high integrity software.

BIOGRAPHIES

John C. Cherniavsky received his Ph.D. in Computer Science from Cornell University in 1972. He is the author of more than 30 papers in software engineering and theoretical computer science. He was on the Computer Science faculty of SUNY Stony Brook from 1972 to 1980. From 1980 to 1984 he was Program Director for Theoretical Computer Science at the National Science Foundation. From 1984 to 1990 he served as Professor and Department Chair at Georgetown University. He has been associated with standards activities at NIST since 1978. Currently he is the Program Director for the CISE Institutional Infrastructure program at the National Science Foundation.

D. Richard Kuhn is a computer scientist at the National Institute of Standards and Technology, where he is involved with operating system interface standards, formal methods, and computer security. He served on IEEE 1003.1 and 1003.2 POSIX committees, is Secretary of IEEE 1201.2, and received a Bronze medal from NIST in 1990 for work on Open Systems standards. He received the M.S. in computer science from the University of Maryland in 1985.

Dolores R. Wallace is currently Project Leader for Software Quality and Safety at the National Institute of Standards and Technology leading an effort in the Assurance of High Integrity Software. She has developed standards and guidelines for Software V&V, Software Acceptance, and Software Management. She supports Federal agencies with software product assurance. Previously she worked for the US Navy in computer applications, especially graphics. She has chaired the COMPASS '90 Conference, served as Co-guest Editor of a special issue of IEEE Software on Software V&V. She is an IEEE seminar lecturer. She received her MS in mathematics from Case Western Reserve University.

HIGH INTEGRITY SOFTWARE STANDARDS ACTIVITIES AT NIST

John C. Cherniavsky, National Science Foundation
Washington, DC 20550

D. Richard Kuhn
Dolores R. Wallace
National Institute of Standards and Technology
Gaithersburg, MD 20899

1. Introduction

The National Institute of Standards and Technology (NIST) has supported the development of standards for software verification and validation and computer security ([1],[2]). Software has become more and more an integral part of our daily lives as it is used in monitoring devices, in controlling devices, and in managing an ever more complex society that is becoming critically dependent upon accurate information. This trend requires better mechanisms to ensure the correct operation of software used in applications requiring high integrity.

In the United States, particular attention has been placed on systems handling classified data, military weapons systems, and nuclear reactor control systems. A number of standards have been put in place that partially address the integrity of these systems ([3],[4]) and some research has begun in the area commonly known as "Software Safety" [5].

In Europe, the coming of a common European market in 1992 has precipitated work in harmonizing disparate standards and in unifying criminal and civil liability codes [6]. These codes now apply to software developers in addition to the Professional Engineers. These codes hold a software developer civilly or criminally liable if software that was supposed to be of high integrity fails. In order to protect against lawsuits developers must be able to show that proper procedures were followed in ensuring the integrity of the software. These proper procedures imply that a standard for developing high integrity software exists (which is not the case). United States firms doing business in Europe will have to conform to these liability laws and will need to use whatever standard that is eventually developed. A generic basis for such a standard will likely involve certification as described in the ISO-9000-9003 quality contractual standards.

Both internationally and within the Common Market, the European Community has active programs for investigating ways of producing high integrity software (through the Esprit program) and in standardizing methods for producing such software. The standardization efforts range from the very specific proposals embodied in MOD-0055 and MOD-0056 [7] to very generic quality standards embodied in the ISO 9000 series of quality standards.

The approach embodied in the ISO 9000 series of quality standards is to develop guidelines for third party certification. The third party certifies that quality controls are put in place by the first party supplier of a product or service to a second party customer. The ISO 9004 standard provides quality guidelines for incorporation into first party production practices. The licensing of the third party certifiers would be accomplished by government or trade entities. Thus a customer requiring software meeting certain integrity conditions would use a supplier who is certified to produce software at the desired integrity level. United States firms marketing their software abroad will either have to use these certification laboratories (which, in some cases, would require revealing trade secrets) or depend on negotiated bilateral treaties where a United States methodology (based upon some standard) could be used to certify the integrity of the software.

NIST has been monitoring the development of European and International standards activities for the assurance of high integrity software. In order to meet its responsibilities

under the Omnibus Trade Act of 1988, NIST has been involved in a number of activities to develop the capability to ensure that software meets integrity constraints. In this paper we report on three such activities: workshops for the development of standards and guidelines for ensuring the integrity of software, the national voluntary laboratory accreditation program, and the development of a formal methods laboratory. In light of these activities, we discuss possible scenarios for the assurance of high integrity software.

2. Workshop Report

With a Workshop on the Assurance of High Integrity Software at NIST on January 22-23, 1991, and a follow-on working group on June 28, NIST took a first step toward involving a broad community of interested parties in the development of guidance for assuring high integrity software. The purpose of this and future workshops is to provide a forum for the discussion of activities that NIST plans to undertake to accomplish this goal. The workshop participants will also be asked to comment on technical contributions as they evolve. The results of these activities will be used to provide information to officials responsible for how the United States should involve itself in international trade agreements relating to software for which high integrity is required.

Participants at the first workshop represented Federal agencies, Canadian government, academia, and industry. The participants were split into four working groups which discussed the following issues:

1. Techniques for developing and assuring high integrity software.
2. A cost-benefit framework for selecting techniques.
3. Controlled and encouraged practices for use in software development.
4. Techniques for criticality assessment and hazard analysis.

The Techniques group's task was to determine a method for describing and identifying techniques for assuring high integrity. The Cost-Benefit group was charged with investigating means for selecting techniques and assurance methods based upon the cost of those techniques and the benefits that accrue from the assurance those techniques can give regarding the integrity of the software. The Hazard Analysis group was asked to to identify both criticality assessment and hazard analysis techniques for high integrity software. The Controlled and Encouraged Practices working group was responsible for studying the forbidden practices of DEF STAN 00-55 [7] and identifying how best to handle them. The working definition of high integrity software was "software that operates exactly as intended without any adverse consequences, including when circumstances outside the software cause other system failures". Examples of requirements for such software come from safety critical applications, security applications, and some commercial applications where the cost of assuring high integrity is balanced by the benefits of the assured higher quality and freedom from the consequences of software failures.

2.1. Techniques Session

2.1.1. Overview

The Techniques working group considered development and verification techniques that can be effective in the production of high integrity systems. A survey of participants showed the following interests and application areas: security, communication protocols, nuclear power systems, weapons systems, formal methods and tools research, railway systems, avionics, independent validation and verification, and quality assurance. There were no Techniques session participants from medical or financial application domains, or from CASE tool vendors, although these are equally relevant areas. A template for describing the characteristics of various techniques (figure 1) was adopted. In addition to

describing features of the various techniques, the template looks at how a technique fits into a development organization by considering the personnel roles involved in its use (e.g., specifier, verifier, coder). Advantages and disadvantages of tools were also considered. To evaluate the effectiveness of the template, small groups were formed to review seven methods considered useful for high integrity systems: Harlan Mills' Cleanroom method; the four formal specification languages EHDM, FDM/Inajo, Estelle and Larch; the Petri-net based tool IDEF0; and "traces", which can be used to formally describe a system by specifying externally observable events.

HOW IT WORKS
Conceptual basis
Representations used
- Text, graphics, etc.
- Executable

Steps performed
- Mechanics - "transform this to that"
- Synthesis and analysis steps
- Tools used

Artifacts produced
- Documents
- Data
- Representations

Roles involved
- Person to task mapping - example: specifier, verifier
- Skills required

WHAT IT ACHIEVES WITH RESPECT TO HIGH-INTEGRITY
Positive
- Errors identified
- Evaluation data produced
- Reuse possibilities

Negative
- Fallibility - common failures, gaps in knowledge, ...
- Bottlenecks - sequential steps, limited resources, skills, ...
- Technical barriers

Other techniques
- Required
- Supported

CURRENT APPLICABILITY OF TECHNIQUE
- Domain of application?
- Where is it being used? How? Where is it taught?
- Who is researching it? Why are they doing this?
- If not in use but has potential, then what changes are needed?
- Maturity:
  Adapt/deal with change? How well does it scale?
  Who can use it? How does it fit with, e.g. prototyping?

Figure 1. Proposed template for describing techniques

2.1.2. Review of Techniques

After completing the templates, some of the techniques were discussed by the full working group, although time did not permit a discussion of all the techniques. The full report on the techniques and their assessment is incorporated in the report on the workshop [8].

2.1.3. Discussion

The tools and techniques discussed have different strengths. All are useful for assurance of high integrity software, although none is comprehensive enough to be used alone. Proper matching of techniques to problems is needed. Application domain, project organization, and personnel skills must also be considered. A high integrity software assurance standard could identify a set of techniques and associate them with the problems that the techniques acceptably address. The participants did not believe that any particular set of techniques should be required for all high integrity software. Technologies are not equally appropriate to all types of applications, so application domain specific standards may be useful. Working group participants sought to make the template categories sufficiently detailed for intelligent selection of techniques, either by developers or for application specific standards.

2.1.4. An Assurance Model

A model of assurance levels, shown in figure 2, was proposed. Working group participants agreed that the proposed model does a good job of structuring assurance levels based on formal methods. No claim is made that increased integrity is guaranteed by higher levels of the model (since the model represents only one axis of a many dimensional problem).

Level	Technique	Examples
3	mechanical verification	ASOS, LOCK, FM8502
2	formal spec + hand proof	VIPER, Leveson's method, traces
1	formal spec only	Z, VDM, control law diagrams
1/2	pseudo-formal spec	STATEMATE
0	static code analysis	SPADE, MALPAS

Figure 2. Assurance levels with formal methods

2.1.5. Recommendations

The group prepared a set of recommendations for inclusion in a standard for high integrity software. The recommendations are necessarily preliminary, but there was a good deal of consensus among participants.

Respect the "practical assurance" limit. With current technology it takes about one year of testing to assure that a system operates correctly for one hour (with a failure probability of one in ten thousand). It was noted that this can be bettered with N-version programs if one assumes the independence of each version.

A standard should state characteristics of techniques and require arguments as to why a technique selected is appropriate. The group felt that techniques are not equally applicable to all application domains. A developer who wishes to claim conformance to a high-integrity software standard will need to describe the characteristics of the application and give a convincing argument as to why the techniques used are appropriate.

A clear implication of this recommendation is that a single all-encompassing standard for high integrity software is not practical unless it is simply a catalog of techniques.

Requirements for specific techniques will need to be based on application domain characteristics. This is in line with the "framework" approach of having a standard that gives general requirements, supplemented by standards at an appropriate level of specificity for different application areas.

Evaluate and track on-going applications of techniques. It is essential to monitor applications of different techniques to determine which are most cost effective for different applications. An equally important aspect of tracking applications is to make techniques more widely known in the industry. Many significant techniques are little used today because practitioners are not aware of them, or because they are perceived as too expensive or impractical. Measuring the costs and benefits associated with various techniques will allow decisions to use techniques to be based on sound data rather than guesswork.

Distinguish between techniques used to eliminate design flaws and techniques used to maintain quality in the presence of physical failures. High integrity systems will require the use of both types of techniques. Determining the optimal tradeoff between fault tolerance and fault elimination for a particular application is a challenging problem. Experience and empirical research will be necessary for designers to make this tradeoff. A standard should provide a selection of both types of techniques, and guidance should consolidate experience to help developers make choices between the techniques.

It was noted that the most important part of a recommendation on techniques is to point out fallibilities. All techniques have limitations; by noting these, developers will be able to compensate for the limitations or at least attach appropriate caveats for purchasers.

It was also recommended that a notation to express what techniques were used at different stages of the lifecycle be developed. Such a notation would facilitate specification of development requirements, and could also be used to characterize developments to make it easier to compare projects.

2.2. Cost-Benefit Session

The Cost-Benefit group addressed the following topics:

- definitions
- model for cost-benefit
- experiment applying a standard
- relationship of cost-benefit to workshop and to a standard.

2.2.1. Definitions

Definitions of the basic terms and concepts need to be established so that the scope and frame of reference for a standard will be clear. The working group has suggested definitions for *cost, benefit, high integrity, software, relevant application domains,* and *users of the standard.* The final standard must identify terms and their definitions that may exist with different definitions in other standards. The definitions appear in the full report.

2.2.2. Model

The Cost-Benefit working group was pessimistic that one standard can satisfy all application domains, development environments, and user environments. The group discussed how to provide users of a standard with sufficient guidance for selecting development and assurance techniques that are affordable and suit the assurance needs of a user. A model was proposed for determining the costs and benefits of techniques for assuring high integrity software. The foundation for this model is described in three papers [9,10,11] The model, shown in figure 3, illustrates how to find the minimum of the cumulative cost of failures per unit time. The working group recommends use of this model

only as a starting point for identifying the parameters that must be built into a selection framework.

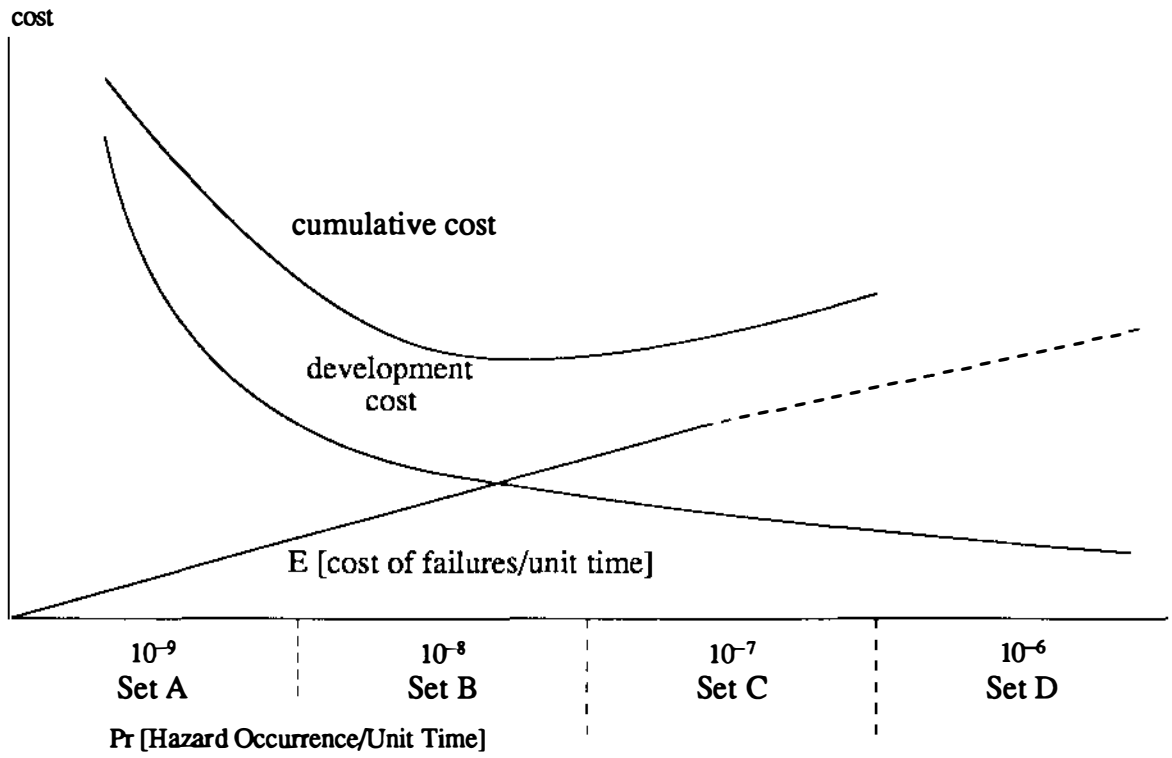


Figure 3. Proposed cost-benefit model for technique selection

A complete model must associate techniques with error types, application domains, and other items relative to a supplier's environment. One suggestion is to associate a probability of failure with a set of techniques as indicated in figure 3. Considerable research is necessary to determine exactly what those techniques are. The original objective was to associate a required level of assurance with a set of techniques. Is a level of assurance identical with a probability of failure per unit time? That is, is all assurance simply a matter of reliability?

Another issue concerns the grouping of methods in general. For example, will techniques highly suited to locating timing errors, a concern of many critical real time systems, be included in Set A? Will others be included in Set B? Extrapolation of this brings up the question: What error classes will be covered by techniques in each of the sets? It may be the case that no obvious clustering of techniques occur because the techniques are best characterized multi-dimensionally and scattering occurs. Can application domains be characterized by their error classes? Does a set of methods then refer equally to application domains and to error classes? Of course, this model focuses on errors, whereas other requirements for assurance may focus on specific qualities (e.g., maintainability, portability, efficiency). It is not immediately obvious that this model, even with sets of techniques, will accommodate selection of techniques based on the qualities they support. How should applicability of any set of methods be described?

Further development of this model requires the collection of data on failures of systems, types of techniques for development and assurance and the errors they prevented or discovered, and the costs associated with the failures and successes. One possible source

of data is the NASA Goddard Space Flight Center's Software Engineering Laboratory [12]. It is not clear if the data collection task should be pursued with the intent that such data would result in a "hard-coded" framework or if the objective should be to lay out a model that users may tailor to their own projects. In this case, users would study data from their environment; that data would have to be of the same type already identified but on a much smaller scale. According to Dr. Victor Basili, due to differences in environments, experiences satisfactory in one environment may become unsatisfactory in another [13]. The working group needs to study how problems such as these will affect generic models.

2.2.3. Experiment

Implementing a standard may mean major changes in the way software is developed and assured. Suppliers may have to provide specialized training for their staffs, and may have to invest in software tools. Training may be needed to help managers understand scheduling for new tasks, new ways of doing traditional tasks. Some of the proposed requirements may be difficult to implement and may not be affordable. The working group strongly recommends that a draft standard should be applied to an industry project. Data from this experiment should influence changes to the draft. The basic structure of the experiment is shown in figure 4.

WHY

- Trial run of the standard to show feasibility
- Acquire performance and cost data on proposed methods

WHAT

- Development according to a draft standard of a realistic sample product in a typical industrial setting
- Measurement of predefined metrics and acquisition of relevant artifacts.

HOW

- NIST, industry, academia form a team
- Find funding
- Prepare strawman draft standard in parallel with planning/preparing experiment

Figure 4. Proposed structure for trial use of draft standards

2.2.4. Session Summary

The Cost-Benefit group strongly recommends that the fundamental terms be defined first, especially software safety. The definitions will define the scope of standards for high integrity software.

The Cost-Benefit group considers the model in figure 3 a starting point for determining selection of techniques. Support of this concept will require the collection of data, much of which may not be easily available. Cost may not be quantifiable or even predictable (i.e., it might be intangible). Other ways of measuring the input to a framework should be considered. The concept of a framework itself implies the development of several standards and guidelines.

Development of standards for high integrity software must also include means of demonstrating conformance to the standards. It must be shown that the requirements of these standards can be met at reasonable cost.

2.3. Controlled and Encouraged Practices Session

This session's charge was to review the history and international standing of practices which have been forbidden or discouraged by some software development standards. Examples of these practices include interrupts, floating point arithmetic, and dynamic memory allocation. A well known example of a standard containing such prohibitions is the British Ministry of Defence DEF-STAN-0055*. The DEF-STAN-0055 prohibitions were based upon the difficulty of assessing code that uses these programming practices, not because the practices themselves are inherently dangerous.

After review and discussion of other standards and their approach to error-prone practices, the group redefined "forbidden practices." The new definition hinged on the concept of "controlled" versus "forbidden" practices. No one believed that all instances of the "forbidden practices" were in fact unsafe, and those that currently are, may be safe in the future if certain technologies develop. This view reflects the comments from the Institution of Electrical Engineers (IEE) and other organizations concerning the DEF-STAN-0055 standard. Other standards discourage but do not prohibit certain practices.

The group adopted this definition of a Controlled Practice:

> A Controlled Practice is a software development procedure which, when used in safety-critical systems in an inappropriate manner, results in an increased level of risk. This practice is one which can reasonably be expected to result in a safety hazard, or, is not demonstrably analyzable.

2.3.1. Encouraged Practices

Certain software practices, although not inherently dangerous, are generally recognized as increasing the incidence of software failure and hence the risk in safety-critical systems. These same practices may be less error prone when certain checks and balances are employed in their use. That is, these "risky" practices inject a certain error type and may only be used in conjunction with other practices which have been shown to detect, mitigate, or counter the error type injected.

Initially the group thought that "encouraged practices" could also be required to offset "controlled practices" in certain circumstances. Later discussion showed that some "good" practices (e.g., use of higher order languages), should be encouraged but not forced or controlled as tightly. Thus, group consensus allowed the change to "Controlled and Encouraged Practices." It was later noted that this would allow developers to choose various combinations of techniques (some familiar to them and others not familiar) as long as the error types were "covered".

2.3.2. Software Integrity versus Controlled and Encouraged Practices

The group decided many factors influenced the risk of the same practice in different domains and applications. A matrix was formed in which such factors would allow a developer to select enough countering practices to allow the use of a "controlled" (not forbidden) practice. This matrix is driven by the level of software integrity required.

---

* At the time of the workshop, the participants had access only to the Draft DEF-STAN-0055, and addressed the "forbidden practices" issue accordingly. The recently-released INTERIM STANDARD DEF-STAN-0055 has relaxed its policies on discouraged or forbidden practices.

The required software integrity level is a result of the hazards identified with the system and allocated to the software (and hardware) (sub)system under design.

2.3.3. International Issues

There was a consensus that the view of "controlled and encouraged practices" expressed in the working group is different from that in the international standards reviewed. Accommodation of this difference requires two definite steps:

1. The definitions of controlled and encouraged practices must map onto all known standards which have such concepts, and
2. The international community must be made aware of the intent of these redefinitions.

Item 2 raises a particularly strong issue because the DEF STAN 0055 forbids practices while a U.S. standard will probably allow practices with certain rigorous development practices. This seems to be diametrically opposed and may preclude mapping one to the other. While the DEF STAN 0055 is a draft standard with comments being addressed, it is being considered as a baseline for European Economic Community (EEC) standards. Thus, NIST should move to promote public discussion of differences and coordinate less strict status on certain practices.

2.3.4. Session Summary

The group advised continuing its own efforts on development of concepts, mapping of *integrity level* to *techniques* and to *controlled/encouraged practices* and mapping to other standards. A list of controlled/encouraged practices should be prepared for consideration by the Techniques group. There must be discussion of unresolved issues (e.g., indirect effects of proposed control of listed activities and indirect hazards of such software as finite element analysis tools used on passenger aircraft or highway bridges). The relevant national/international organizations developing related or similar standards need to be contacted.

2.4. Hazard Analysis Session

The working group on Hazard Analysis had the task of defining the terms and techniques for hazard identification and analysis of software for which assurance of high integrity is desired. Experts from both the military and civilian sector were present. Two documents, MIL-STD-882B [3] and the "Orange Book" [4], were used as initial examples of the activities that might be present in dealing with hazard analysis. Although the Orange Book does not directly address hazards, the fact that it describes assurance levels for certain types of potential security breaches makes it relevant because, as mentioned in DEF-STAN-0055 [7], security breaches can be viewed as hazards. The initial objective of the session was to identify techniques for:

1. identifying hazards,
2. classifying hazards,
3. identifying critical systems,
4. determining how much analysis is necessary,
5. determining where to do analysis, and
6. conducting trade-offs.

In order to accomplish this objective, agreement was needed on many of the terms. In particular, the terms *hazard, risk, and criticality* all needed definition in the context of high integrity software. Analogies were drawn from a number of areas in pursuit of

common definitions to apply to high integrity software. From the system safety area hazards and risks are well defined in MIL-STD-882B. The events to be avoided are injury and death; the hazards are elements of the environment that can cause these events. From the perspective of a mission during wartime the events to be avoided are the inability to fulfill a mission and the hazards are elements of the mission environment that can cause these events. >From the perspective of security the events to be avoided are security breaches and the hazards are elements of the environment whose presence or absence can allow these events to occur. From the perspective of the manufacturer of consumer products containing embedded software, the events to be avoided are losses caused by deficiencies in the product that result in financial losses and the hazards are elements of the products environment that could allow these events to occur. These events could be as simple as a ROM error in a chip in a washing machine requiring a recall costing $100/machine. The events to be avoided are called mishaps.

2.4.1. Basic Notions of Hazard Analysis

Given the wide variation in the events to be avoided, the following definitions of mishap and hazard were adopted.

Mishap - An unintended event that causes an undesirable outcome.

Hazard - A condition that may lead to a mishap.

Given this definition of hazard, the notion of risk is defined to be a function of the hazard, the vulnerability to the hazard, and the opportunity for the associated mishap to occur. The vulnerability and opportunity are assessed together to obtain a probability of the mishap occurring. Once a rough probability has been obtained, decisions are made as to the criticality of the hazards in order to determine whether actions are necessary to mitigate the hazard (or if the consequences are sufficiently severe, not to build the system). As an example, consider a nuclear power reactor and the hazards posed by meteor strikes and earthquakes. The vulnerability of the reactor to a meteor strike is high while the opportunity of the mishap occurring is very low. Thus actions aren't taken to mitigate the hazard (reactors aren't built under a mile of rock) even though the consequences of reactor failure are severe. The vulnerability of a reactor to an earthquake is high and the opportunity for occurrence, particularly on fault lines, is sufficiently high with consequences sufficiently severe (a function of the hazard) that actions are taken, such as building away from fault lines, to mitigate the hazard.

One method of performing this analysis involves building a hazard criticality chart, as illustrated below in figure 5. The method is described further in MIL-STD-882B.

The letters A-E stand for probabilities of a particular hazard resulting in a mishap. A is the most frequently occurring (nominally "frequent") while E is the least frequently occurring (nominally "improbable"). The Roman numerals I through IV represent the severity of a mishap caused by the hazard. For the safety concerns of MIL-STD-882B, I stands for death or system loss, II stands for a severe occupational illness or major system damage, III stands for minor injury or minor system damage, IV is negligible injury or damage. The regions labeled 1 to 4 are determined by policy. In MIL-STD-882B region 1 is unacceptable, region 2 is undesirable, region 3 is acceptable with approval, and region 4 is acceptable. For each hazard, determined by a careful analysis of the environment in which the system is operating, such a chart is drawn up. Values for the probabilities of a hazard occurring and the severity of a mishap arising from the hazard are determined. If these are in the unacceptable or undesirable range then steps must be taken to mitigate the severity of the mishap and/or reduce the probability of the hazard resulting in a mishap.

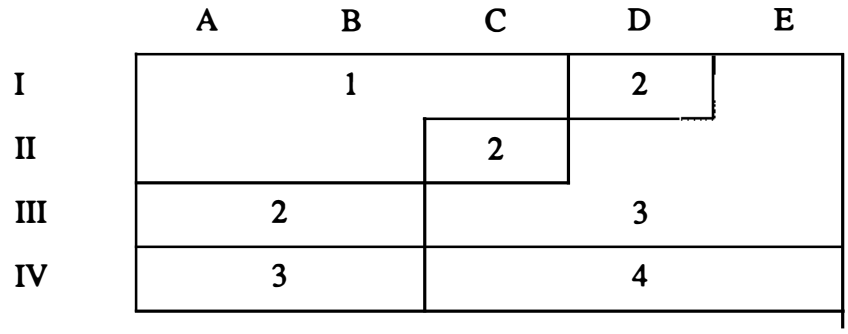


Figure 5. Hazard Criticality Chart

This same hazard analysis chart can be adapted to high integrity domains outside of safety. What needs to be identified are the roman numeral categories I to IV. For example, to analyze hazards for military missions I would correspond to an inability to fulfill the primary mission capabilities (e.g., field an army in a war zone), II would correspond to an inability to fulfill a secondary mission (e.g., an impaired offensive capability against a collection of targets), III would correspond to an inability to fulfill support functions, and IV would correspond to an inability to fulfill administrative functions. In a consumer product the categories I to IV could correspond to I: product causes death or injury, II: product causes damage resulting in financial loss to consumer, III: product does not perform its function resulting in financial loss to company, and IV: product does not satisfy a few consumers in ways unrelated to functionality, cost, or death or injury.

It is possible that five probability levels and four mishap severities do not result in a good model. One may want to refine or coarsen the granularity of these categories. The principle underlying constructing such modified hazard analysis charts can be extrapolated from the above examples. It must be emphasized that not only must the chart be carefully constructed, but also the policy of accepting or rejecting the hazards (labeling the regions) must be clearly articulated.

Techniques for identifying and classifying hazards and for determining the risk associated with hazards is domain dependent. Generic methods include "lessons learned" (historical information about previous mishaps), analysis of energy barriers and the tracing of energy flows (mishaps are frequently associated with energy release or containment), previous system analyses, adverse environment scenarios, general engineering experience, and tiger team attacks (essentially brainstorming).

Specific techniques in tracing possible effects of hazards and isolating those effects have been developed over the last 40 years. These include fault tree analysis, failure modes, effects and criticality analysis, event tree analysis, and hazard and operability studies. At the code level formal proof of correctness and various data and control flow analyses can be performed [5]. Isolating parts of the system responsible for assuring high integrity is an important method of limiting the complexity of the analysis necessary for assurance. This is exemplified in the notion of a Trusted Computer Base that is integral to the TCSEC ("Orange Book") [4] standards. The design techniques described in the TCSEC include the isolation of critical functions in kernels, assurance of module independence ideally through referential transparency of modules, and the general isolation from access of critical software and data.

2.4.2. Lifecycle Hazard Analysis Activities

An assessment of hazard criticality is required during all phases of software development. Hazards to be avoided or mitigated must be identified. This implies that there is a strong traceability requirement for hazards that must be avoided or mitigated. This also implies documentation requirements at all lifecycle phases for hazards being traced. Modifying the language of the software systems safety community, it is necessary to have a Software Integrity Preliminary Plan, Software Integrity Subsystem Plans, Software Integrity Integration Plans, etc. as described in MIL-STD-882B and in the workshop report.

Throughout this process, standard software quality assurance activities are followed. Quality assurance is a prerequisite for high integrity software. Assurance includes checking that the software addresses the hazards, and developing tests that "exercise" the software in response to external events that may lead to a hazard (i.e., ensuring that all hazards are "covered"). This testing requires (as does traceability, etc.) isolating the software that addresses hazards. This isolation also allows for more intensive validation activities, such as the use of formal specification or even the formal proof of high integrity properties, to be used.

3. National Voluntary Laboratory Accreditation Program

Standards provide a means of specifying requirements for software development and acquisition. A standard will make it possible for industry and government organizations to have uniform quality requirements for high integrity software. But a standard provides little benefit without a means of ensuring conformance to the standard. Developers and other organizations have been responsible for verifying compliance to a standard. Because the cost of showing conformance may be prohibitive for purchasers to conduct conformance tests on their own, laboratories have been set up to show conformance. In the electrical products industry, Underwriters Laboratories tests products for conformance to standards. Buyers can have confidence that a product with U.L. approval meets a minimal safety and quality standard.

The European community is now establishing software testing and certification laboratories. In the U.S., various government and industry organizations have begun similar efforts for their areas of concern. The FDA has established software testing criteria for medical device software; the National Computer Security Center funds evaluations of secure systems developed to the requirements of the "Orange Book."

In its role as the nation's measurement and test laboratory, NIST established the National Voluntary Laboratory Accreditation Program (NVLAP) in 1976. The objective of NVLAP is to improve the efficiency and reduce the cost of conformance testing by accrediting testing laboratories. Vendors can submit products that claim conformance to a standard to a NVLAP accredited lab. The lab conducts tests and evaluations of the product, then submits the results to NIST for certification. The NIST certification provides a minimal assurance to purchasers that the product conforms to the standard.

Although originally intended for accrediting labs that do testing of chemicals, metals, and other physical products, NVLAP has been extended to the software industry. Laboratories for testing conformance to the government's Open Systems Interconnect (OSI), GOSIP, were accredited in 1990. POSIX operating system interface conformance testing labs are being accredited in 1991. NVLAP conducts a thorough evaluation of a lab's personnel and equipment to ensure that the lab is competent to test conformance to a particular standard. Once a high integrity software standard is in place, NVLAP accredited laboratories could provide a cost-effective means of ensuring that software products conform to the standard.

4. A Formal Methods Laboratory

Formal description methods have been suggested as techniques for producing high quality software. NIST has begun evaluation of a number of systems supporting formal methods for the development of software. These systems include nuPrl, FDM, EHDM, and Estelle. There are three aspects to this work: evaluation of formal methods and tools through use in "real world" software development projects; construction of additional tools; and the specification of standards in a formal language.

Previous projects include formal verification of the design for a smartcard-based access control system [14], the specification of a message authentication device [14], and the formal specification of the ISO Transaction Protocol (ISO 10026). To assist in evaluating software for conformance to security standards, a suite of static analysis tools has been developed [15], and a program slicing tool is currently being completed [16].

The fundamental mission of NIST has always been to develop methods for the precise specification of standards, and to support precision measurement and test procedures used to show conformance to standards. With few exceptions, the semantic content of software standards is expressed in natural language (although the syntax may be defined formally.) This typically results in ambiguities that lead to incompatibilities among implementations from different vendors, and inconsistencies between test suites developed by different organizations. To ensure that implementations are consistent and conformance tests are equivalent, standards must be expressed in an unambiguous notation. This point has been recognized by others [17]. An effort in this area is the specification of the Federal Information Processing Standard 140-1, "Security Requirements for Cryptographic Modules" [18]. Some standards are deliberately defined to have some ambiguities. In these cases, a formal specification may be harmful. The above comments apply to standards in which precision overrides flexibility.

An ultimate goal of this work is to develop two capabilities. The first is the capability to express standards in a formal language. Actual implementations of systems can then be validated using the formal specification and tools available from systems such as FDM. The second is to evaluate these systems and make recommendations regarding their appropriateness, cost, effectiveness, strengths and weaknesses for developing high integrity software.

5. References

[1] "Guidelines for Lifecycle Validation, Verification, and Testing of Computer Software," FIPSPUB101, National Bureau of Standards, Gaithersburg, MD 20899, 1983.

[2] "Data Encryption Standard," FIPSPUB 46-1, National Bureau of Standards, Gaithersburg, MD 20899, 1988.

[3] MIL STD 882B, Task 308, Software Safety Requirements Traceability Matrix and Analysis - DOD HBK-SWS, 20 April 1988, Software System Safety, Department of Defense.

[4] US DOD Trusted Computer System Evaluation Criteria, DOD 5200.28.STD.

[5] Leveson, Nancy, "Software Safety," Computing Surveys.

[6] Robertson, Ranald, "Product Liability in the UK - Issues for Developers of Safety Critical Software", COMPASS 90, IEEE, pp 178-181.

[7] Draft Interim Defence Standards 00-55 and 00-56, Requirements for the Procurement of Safety Critical Software in Defence Equipment, Requirements for the Analysis of Safety Critical Hazards, Ministry of Defence, Room 5150A, Kentigern House 85 Brown Street, Glasgow G2 8EX, May 1989.

[8] Wallace, Dolores R., D. Richards Kuhn, and John C. Cherniavsky, "Proceedings of the Workshop on High Integrity Software; Gaithersburg, MD; Jan.22-23, 1991", NIST SP 500-190, National Institute of Standards and Technology, Gaithersburg, MD, 1991.

[9] Hecht, Herbert, "Figure of Merit for Fault-Tolerant Space Computers," IEEE Transactions on Computers, Vol C-22, No. 3, March 1973, pp 246-251.

[10] Hecht, Herbert, "Allocation of Resources for Software Reliability," Proceedings COMPCON Fall 1981, IEEE, 1981.

[11] Hecht, Herbert, "Effectiveness Measures for Distributed Systems, " Symposium on Reliability of Distributed Software and Database Systems, IEEE, 1981.

[12] Humphrey, W.S., et. al., "A Method for Assessing the Software Engineering Capabilities of Contractors," CMU/SEI-87-TR-23, Software Engineering Institute, Pittsburgh, PA, 1987.

[13] Basili, Victor R., "Software Development: A Paradigm for the Future," Proceedings 13th Annual International Computer Software and Applications Conference (COMPSAC), Orlando, FL, IEEE, September, 1989.

[14] Kuhn, D. Richard and James F. Dray, "Formal Specification and Verification of Control Software for Cryptographic Equipment," 6th Annual Computer Security Applications Conference, IEEE Press, 1990.

[15] Kuhn, D. Richard, "Static Analysis Tools for Software Security Certification," 11th National Computer Security Conference, NSA/NIST, 1988.

[16] Lyle, James and Keith Gallagher, "Using Program Slicing in Software Maintenance," IEEE Transactions on Software Engineering, Aug. 1991.

[17] Blyth, D., C. Boldyreff, C. Ruggles, and N. Tetteh-Lartey, "The Case for Formal Methods in Standards," IEEE Software, Vol. 7, No. 5 (Sept. 1990).

[18] "Security Requirements for Cryptographic Modules," FIPS 140-1 (draft, 1990), National Institute of Stanrds and Technology, Gaithersburg, Md. 20899.

Stephen A. Bender, CQA

Steve Bender, President of The Quality Connection, is a management consultant and veteran in the Quality Assurance field. He has participated in areas ranging from operations, programming, analysis, and project direction to training management, workflow simplification, professional facilitation, management development, and executive staff. His experience in psychology issues relevant to the technical workplace is extensive. With significant experience in Total Quality Management, and over 20 years experience in the data processing field, his interactive style is well known to thousands of attendees.

He has been an active seminar and workshop leader and keynote speaker for the Computer Security Institute, the Work in America Institute, and the Quality Assurance Institute, State and Federal Governments, and numerous personal, professional, and business organizations and corporations. He actively teaches and consults in the principles of Total Quality Management (TQM) before a number of Fortune 500 companies, including technical quality awareness and tools training, and human resource development (culture change) for TQM, including preparation for the Baldrige Quality Award.

He has edited and contributed material for books on testing, standards, human resources, and on computer assessment of mutual funds, and written for *Government and Computer news,* the *QAI Journal*, *Industry Week*, the *Capital District Business Review, Auerbach* and others. He is author of *Total Quality*, a nationally advertised Audio Tape series on quality of personal and professional life (seen on CNN, Lifeline, and Channel America).

Steve chairs the PEOPLE track for the Quality Assurance Institute's National Conference, and is also a member of their Board of Advisors. He is a Certified Quality Analyst, and a Certified Master Practitioner and Trainer of Neuro-Linguistic Programming (NLP), the world's most effective process for communication and behavior/culture change. To our knowledge, he is the only one in the world jointly certified as both a Trainer of NLP and a Certified Quality Analyst. In addition to his Quality Assurance, consulting, and teaching roles, he is an active counselor using NLP.

Most recently, Steve has conducted a workshop on Quality in Zermatt, Switzerland before members of 15 countries, has written for *Productivity SA* of South Africa, and has been invited there to improve corporate quality for those wishing to end Apartheid. Although unable to attend, he had been selected to join a delegation of Americans to discuss quality before the Soviet Union and Belgium, as part of the Citizen's Ambassador Program chaired by President Reagan. In addition, he as recently chaired the *Total Quality for Financial Services Excellence* conference in New York City.

Making Quality Come Alive: Getting it to Stick

Outline

I. Pockets of Quality

   A. Bottom up involvement alone - frustrating

   B. Top down involvement alone - this too will pass

   C. Proper role of the advocate and sponsor

II. Training in Quality that Sticks

   A. Content

      1. Costs of Failure

      2. Process vs. Product

   B. Delivery

      1. Personal examples in products and service

      2. Everyday examples and metaphors

III. Practicing Results that Stick

   A. Congruency - walk the talk - don't reward failure

   B. Fred Smith's 3 principles

   C. Empowering individuals - trust

IV. Measuring Results with Authority

   A. Right message, wrong person?

   B. Contributors and Results - "measuring keystrokes"

V. Selling Benefits Internally

   A. What sells your teenager?

   B. Active participation and ownership

   C. Walk in the other's shoes

I. Pockets of Quality

It is not unusual for quality programs to exist here and there in an organization, but not everywhere. Total Quality Management implies having it not only in all areas in the physical plant, but also all areas functionally: products, services, external customers, internal customers, attitudes, behaviors, and interplay among individuals.

When only pockets of quality exist, it is often due to bottom-up involvement and ownership, and it can only go up so far. This frustrates individuals, and gives them the impression that others don't care about quality. When started only at the top, it may also be viewed as "just another management program", and "this too will pass." When taken together, it will happen. Unlike management principles, which show us good managers are made, not born, quality principles in most people are born, then "unmade." Have you ever seen anyone doing a really low quality hobby?

At any given level, there may be an enthusiastic sponsor or advocate for quality. Recent studies show that the more visible the sponsor, the more he/she takes credit for the successes, the more likely it is to fail. Getting buy-in from those who resist change depends upon avoiding the "star" syndrome, even when it is offered. When a critical mass of people move behind quality principles, it becomes institutionalized. Otherwise, those who adapt late may push the effort back and cancel it.

II. Training in Quality that Sticks

Before quality can happen totally in an organization, people have to know what they are doing, and have to want to work there. The first part is training, the second is willingness and attitude.

People have to have training in quality practices in order to make it work. In addition to knowing how to do their job well, this includes:

1. Quality Tools
2. Quality Principles and Awareness

Many quality training programs spend an unbalanced amount of time heavily on the tools (Parieto Charts, Fishbone Diagrams, etc.) rather than the principles (Belief Systems).

Good quality training programs also need to consider:

1. Content
2. Delivery

Many are heavy in the content, but because the delivery style does not achieve buy-in and belief, it becomes something that is learned rather than something that is practiced. This helps along the saying, "this too will pass."

How do we emphasize the proper content on principles as well as tools, while paying attention to method of delivery to get it to stick?

In content, place heavy emphasis on Cost of Quality concepts, as

well as Process vs. Product concepts. These are the things which make the difference in believability. Cost of Quality differentiates huge savings by doing it right the first time, compared to huge costs of letting a failing product or service go out in the marketplace. Process vs. product describes how if you pay attention to the process by which you build a product or service, the product will take care of itself.

In delivery, connect a person's work experience to their own personal examples of grief and joy with a product or service - after all, many aspects of their work life are just like that. In addition, by putting quality in everyday terms and examples - sometimes called "metaphors", you will relate to an individual's belief systems much better. It is these belief systems which describe a person's perception of capability, and then how they behave.

III. Practicing Results that Stick

Well, now that we have delivered the belief system along with the behavioral rules, we have quality that sticks, right?

Not always!

Like a car which runs on gas, we can only go so far on a tankful - before we must seek outside support, reinforcement, assistance (gas station). As Leaders, we must first be congruent - "walk the talk"

- and practice what we preach. Second, we must understand the 3 things people are looking for in their jobs. Third, we must practice empowerment of people in their jobs, and couple this with real trust.

First, we must walk the talk. If we are unable to practice it, we must not preach it. Although most think they are congruent, there may be many accidental, subtle ways in which this is not happening. For example, many accidentally reward failure, by paying more attention to those that "save the day" rather than prevention. What about those who made the day "unnecessary to save" - do they get equal or greater credit?

Second, Fred Smith, CEO of Federal Express, summarized 3 things he felt all employees should have clear answers to:

1. What's in it for me?
2. What's expected of me?
3. Where do I go with a question?

By insuring that all individuals were easily able to answer these questions, he accurately felt that impediments to quality work were removed. If you feel that quality is a natural tendency, unless there are motivations against it, you find these barriers to quality work and eliminate them.

Third, much has been said about empowerment in the workplace. A recent Industry Week (May 4, 1991) article discussed the common fallacy of empowerment - not in its concept, but the way it was

generally practiced. Simply delegating responsibility downward, without the resources and authority, become merely a way of shifting blame. You can tell that empowerment is working, when anyone dealing with a customer is able to make the product or service right without contacting their supervisor.

IV. Measuring Results with Authority

Getting more and more "pockets" - and finally the entire organization - to practice quality requires measurement. Those who are already practicing it can be convincers to those who are not, and only through measurement. There are two issues: right message - wrong person, and measuring contributors.

If you "just talk" about your successes, you are not likely to be convincing. Perhaps worse, if you speak the wrong language, you won't be heard at all. This means speaking technical language to a higher executive will be less meaningful than converting those labor savings into dollars, which is the executive's language.

By internally measuring contributors to quality, rather than results alone, it is less likely that the results will be sabotaged. There are many ways of artificially achieving a desired result in a product through false measures, but few ways of twisting contributing process measures for those products. When there are accidental rewards for poor quality, this becomes even more noticeable.

For example, it is well known that an excellent key entry operator delivers more keystrokes per hour than the rest of their constituency. But what happens when you start to measure this result? Key entry operators have actually discarded or delayed more lucrative batches of work in order to increase their keystroke count. By measuring contributors, such as error rate, entry techniques, and so forth, you will get the results you are looking for, with no subterfuge.

V. Selling Benefits Internally

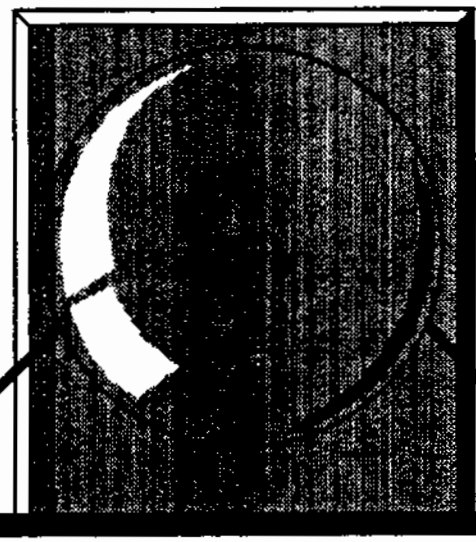
Practicing, and measuring, are excellent selling techniques to spread quality pockets throughout the organization. Active ownership, coupled with "walking in the other's shoes" are dramatically effective at getting buy-in and having others sell themselves, which is always more effective than you selling them.

Those who have raised teenagers know how hard it is to be convincing - yet those same young people who go out on their own are "amazed at what you have learned while they were gone." What causes this? Ownership! Once you experience directly the topic being discussed, it has much greater meaning. Perhaps teenagers are just like us in this area. In fact, are they a bit more honest or direct about it?

For this reason, commitment can be shallow compared to involvement. Commitment may change with shifting pressures, but a person who has

been involved is less likely to shift. The personal experience and investment - indeed, the "emotional sunk cost" - carries tremendous weight.

Finally, Fred Smith's "What's In It For Me" principle shows us the importance of looking at our problems and solutions from the other's point of view. While this seems obvious, even trite, consider this: When you approach someone, are you thinking of what you are going to say, or the way in which you will be received? Think about it!



Making Quality Come Alive:

Getting it to Stick

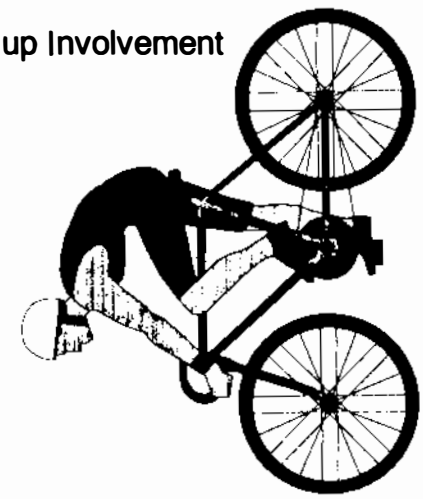
Getting it to Stick

- Pockets of Quality
- Training in Quality
- Practicing Results
- Measuring Results
- Selling Benefits

Management

Total Quality

Bottom up Involvement



Top Down Involvement

Born, not Made?

Made, not Born?

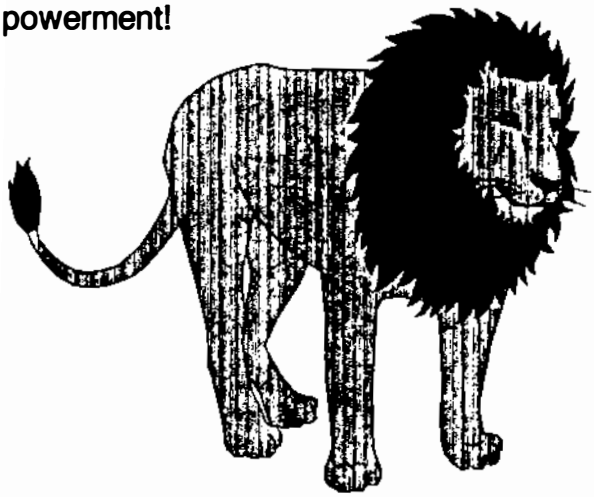
Content:
- Cost of Failure
- Process

Delivery:
- Real examples

The Role of the Advocate

Walking the Talk

Empowerment!



Empowerment:

- T otal
- R esponsibility
- U nderstanding
- S tandards
- T echnology

Contributors → Results

Ownership:

Have you ever tried to teach your teenager anything -

about anything?

SQA STANDARDS AND TOTAL QUALITY MANAGEMENT

Caroline E. Wardle, National Science Foundation *
Dolores R. Wallace, National Institute of Standards and Technology
Reza Khorramshahgol, American University
Eugene G. McGuire, American University
Bonnie Kaplan, American University

ABSTRACT

Total Quality Management (TQM) programs, which focus on continuous process improvement, are becoming increasingly popular in corporations today and may, in fact, be superceding quality assurance activities. Quality assurance has been used by corporations to ensure the quality of released products. What we are finding now is that the thrust of management attention is on process improvement rather than on quality assurance.

This paper examines some major standards and programs in SQA and TQM. Because few companies are using SQA standards, the difficulties in implementing SQA standards are addressed. Results from an on-going research project studying SQA standards and the interrelationships between TQM and SQA are presented.

---

* Dr. Caroline Wardle
National Science Foundation,
CISE/CDA
1800 G Street, NW, Room 436
Washington, DC 20550
Email: cwardle@note.nsf.gov

BIOGRAPHIES

Caroline E. Wardle is the Program Director, Educational Infrastructure, for the Computer and Information Science and Engineering Directorate at the National Science Foundation. Previously she was an Associate Professor of Computer Science at Boston University where she established and chaired the department of Computer Science at Metropolitan College. She was also the Dean of the Wang Institute where she established the School of Information Technology and implemented its first degree program, a Masters degree in Software Engineering. Her research interests have ranged over broad areas in software engineering, computer science and information systems. Her current research work includes software quality assurance, and total quality management in software development.

Dolores R. Wallace is Project Leader for Software Quality and Safety, National Institute of Standards and Technology, and leads an effort in the Assurance of High Integrity Software. She has developed standards and guidelines for Software V&V, and supports federal agencies with software product assurance. Previously Ms. Wallace worked for the US Navy in computer applications, especially graphics. She has chaired the COMPASS '90 Conference and served as Co-Guest Editor of a special issue of IEEE Software on Software V&V.

Reza Khorramshahgol is an Assistant Professor of Computer Science and Information Systems at American University. Previously he was an Assistant Professor at North Carolina's Central University, and before that a Member of the Technical Staff at AT&T Laboratories. He serves on the editorial board of the IEEE Transactions on Engineering Management. His current research interests include software quality management, telecommunications, and decision support systems.

Eugene G. McGuire is an Assistant Professor of Computer Science and Information Systems at American University. Previously he worked in all phases of systems development at Tandem Computers and before that at NetExpress Communications. His current research interests include organizational aspects of systems development and implementation, group dynamics in information systems environments, and technology transfer issues.

Bonnie Kaplan is an Assistant Professor of Computer Science and Information Systems at American University. Previously she was an Assistant Professor of Information Systems at the University of Cincinnati. Her research specialties concern social aspects of computing, impact of computing technologies, and acceptance and diffusion of medical computer applications.

SQA STANDARDS AND TOTAL QUALITY MANAGEMENT

Caroline E. Wardle, National Science Foundation *
Dolores R. Wallace, National Institute of Standards and Technology
Reza Khorramshahgol, American University
Eugene G. McGuire, American University
Bonnie Kaplan, American University

INTRODUCTION

Quality Assurance has traditionally been used by corporations to ensure the quality of released products. When we examine the software development environment of private companies and Federal agencies, we find that few of these organizations are using Software Quality Assurance (SQA) standards, although many have expressed a desire to do so. Even those companies that are using SQA standards are not entirely happy or satisfied.

Another vehicle for improving product quality, Total Quality Management (TQM), is becoming increasingly popular in corporations today. TQM focuses on continuous process improvement and is receiving significant acceptance at the senior management level. What we are finding is that the thrust of management attention is on process improvement rather than on quality assurance.

A research project was initiated in 1990 to examine the use of SQA standards, the adaptation of TQM to software development, and the interrelationships between TQM and SQA. The initial part of the study included data gathered from interviews of SQA personnel and TQM personnel in both industry and Federal agencies. Preliminary findings from this study and findings based on our experience in aiding others responsible for SQA in their organizations, demonstrate some of the problem areas in SQA standards implementation as well as the effect that TQM may have on SQA activities.

In the following sections, we will first present an overview of three major SQA standards and a discussion of the difficulties of implementing these standards. Then we will present an overview of three major TQM programs and address the issues raised in the research study about the relationships between SQA and TQM.

---

* Dr. Wardle is a Guest Researcher at the National Institute of Technology and Standards

1. OVERVIEW OF SQA STANDARDS

The standards related to software quality assurance that will be discussed in this paper have significant differences that managers should understand before implementing them in their organizations. Some of these differences concern the sponsoring organizations and intended audience, the procedures by which the standards were developed, the purpose of the standards, the scope (software development and/or quality assurance), ability to tailor the standards, and customer/developer responsibilities. Figure 1 presents the standards referred to in this paper.

1.1 Sponsoring Organizations, Intended Audience and Standards' Development Procedures

The standards used in this study were developed by different organizations. The Joint Logistics Commanders (JLC) of the United States Army, Navy, and Air Force commands of the Department of Defense (DOD) developed standards which have been adopted by the DOD. The other standards were sponsored by the Institute for Electrical and Electronics Engineers (IEEE), and the International Organization for Standards (ISO).

1.1.1 DOD 2167A, 2168

The JLC undertook the task of developing standards for software development and quality. The US defense industry participated in this activity through the Council of Defense Space and Industries Association (CODSIA) and the National Securities Industry Association (NSIA). The standards were issued as DOD standards and are required on DOD contracts for mission critical computer resources.

1.1.2 IEEE 730

The IEEE is a professional organization which is a standards developing organization accredited by the American National Standards Institute (ANSI). The standards activities for software engineering are sponsored by the IEEE Computer Society Software Engineering Subcommittee. Participation in IEEE standards working groups is open to industry, government, and academia. IEEE standards are often submitted for review and approval as ANSI standards. The standards are used on a voluntary basis.

1.1.3 ISO 9000

The ISO is a formal organization for standards where membership is by country.

DOD-STD-2167A: Military Standard, Defense System Software Development Std, February 29, 1988 US Department of Defense.

DOD-STD-2168: Military Standard Defense System Software Quality Program Std, April 29, 1988, US Department of Defense.

IEEE Std.730.1: Standards for Software Quality Assurance Plans (SQAP), 1989.

ISO 9000 Series: "Quality management and quality assurance standards guidelines for selection and use," 1987.

- **ISO 9001:** "Quality systems - Model for quality assurance in design/ development, production, installation, and servicing," 1987.
- **ISO 9000-3:** "Quality Systems - Guidelines for the application of ISO 9001 to the development, supply, and maintenance of software," 1990.
- **ISO 9004:** "Quality management and quality system elements -- guidelines, 1987.

Figure 1 Standards Related to SQA * [1]

Within the ISO, a national body or other liaison organization may propose one of its standards for adoption. A subcommittee within the ISO may also form a working group to develop a standard or modify one that has been proposed. The working group usually has members from several countries and must follow formal procedures once a document reaches draft stage. To become an ISO standard, a document must pass several levels of balloting, with one vote per country. This is different from the IEEE procedures in which individuals vote on the ballot version of a proposed standard. In both cases, negative ballots must be resolved. The ISO itself does not require use of its standards. Instead these standards are often mandated in various formal trade agreements, in industry agreements or by a regulatory agency.

Several European nations are planning to unify as an "internal market" by the end of 1992 [2]. This European Community (EC 92) is establishing systems for product certification and quality systems registration. Companies wishing to sell EC-regulated products will have to meet requirements of the EC standards. It is expected that ISO 9000 will be adopted for use in the EC 92.

---

* In this paper we will use the abbreviations: DOD 2167A, 2168 for DOD-STD-2167A, 2168; IEEE 730 for IEEE Std.730.1; ISO 9000 for ISO 9000 Series.

1.2 Purpose of Each Standard

1.2.1 DOD 2167A, 2168

These two standards are intended to provide the DOD with the capability to develop and assess quality software. Specifically, DOD 2167A establishes uniform requirements for software development that are applicable throughout the system life cycle. The requirements provide a basis for government visibility into a contractor's development, testing and evaluation efforts. The DOD 2168 standard establishes requirements for a software quality program to be applied during acquisition, development, and support of software systems. The standards are intended to be used together to ensure that the contractor develops software according to a uniform set of requirements for both software development and software quality.

1.2.2 IEEE 730

The purpose of the IEEE's SQAP standard is to provide uniform, minimum requirements for the preparation and content of Software Quality Assurance Plans. It applies to the development and maintenance of critical software.

1.2.3 ISO 9000

The ISO 9000 standard establishes the framework for requirements for quality products of all types, not just software. Its purpose is to clarify distinctions and interrelationships among the principal quality concepts and to provide guidelines for the selection and use of ISO standards on quality systems. Other standards in the 9000 series are for internal quality management purposes or for external quality assurance purposes. The ISO 9000-3 standard provides requirements for the application of ISO 9000 to software.

1.3 Scope: Software Development and/or Quality Assurance

1.3.1 DOD 2167A, 2168

The DOD standards provide requirements for both development and quality assurance. The DOD 2167A standard provides detailed requirements for the contractor's documentation, the 17 documents that must be produced, their format, and the topics that must be included. The standard does not require a specific set of software engineering methods, but does require that selected methods are systematic and well documented.

The general requirements of DOD 2167A provide for software development management, software engineering, formal qualification testing, software product evaluation, software configuration management, and the transition to software support. The standard specifies the products to be evaluated and specifies evaluation criteria for each product. For example, during software requirements analysis, the software requirements specifications and interface requirements specifications are evaluated for internal consistency, consistency with other documents, traceability, appropriate software engineering techniques, appropriate allocation of sizing and timing resources, and adequate test coverage of requirements.

Within DOD 2168, the contractor establishes the software quality program but the contracting agency reviews and approves the program. General requirements address software quality program documentation, planning, implementation, evaluations, evaluation records, corrective action, and management review. The contractor must make available for review, documented evidence that the software, documentation, and activities have met contractual requirements.

While the DOD standards do not specify how to do development and quality assurance, nonetheless, the standards have considerable detail concerning the processes and documentation of development and for the activities of SQA.

1.3.2 IEEE 730

In contrast to the DOD standards requirements which address both development and quality assurance, the IEEE SQAP standard addresses only quality assurance requirements and indirectly includes development requirements.

The IEEE SQAP standard requires that a SQAP contain sections for the following topics: documentation; standards, practices and conventions; software configuration management; reviews and audits; testing; tools and techniques; code control; media control; supplier (subcontractor) control; records collection; training; and risk management. Any organization or groups within an organization can use this standard. The standard can be used for any, or portions of any, life cycle.

The SQAP must identify the documentation for the development, verification and validation, use, and maintenance of the software, and must identify how the documentation will be checked for adequacy. The standard specifies minimum requirements for the software requirements specifications, software design description, software verification and validation plan (SVVP), software verification and validation report, user documentation, and the software configuration management plan. Unlike the DOD standards, the SQAP standard

may not always be used in an environment where its implementation mandates the development requirements.

The SQAP must state what reviews and audits must be conducted and how they are to be accomplished. Testing may be performed under many arrangements, so the SQAP standard requires a section of the SQAP to identify tests not included in the SVVP.

1.3.3 ISO 9000

The ISO 9000 requirements are general. In ISO 9001 and 9000-3, the details of the standard provide requirements for the supplier regarding the framework of the quality system (management, documentation of the quality system, audits, corrective action), the quality system life cycle activities and the quality system supporting activities.

The life cycle activities require a development plan to specify the development phases, required inputs and outputs, and verification procedures, but does not specify documents or procedures. Requirements for design, implementation, review and testing are general. Configuration management, document control, quality records, measurements, and training are some of the supporting activities. For both acceptance and maintenance, requirements are more extensive than those in the DOD and IEEE standards.

The ISO 9004 considers the total quality management system, addressing topics such as economics, marketing, and production in addition to the usual topics like management, engineering, and assurance. It requires a company to examine itself and its ability to plan its product line as related to its quality system and economics.

1.4 Ability to Tailor Standards

Both the DOD and IEEE standards apply to the development and maintenance of critical or high-integrity software. High-integrity software controls services that affect life, property, the national defense, and services that are critical for companies to function in highly competitive business environments. ISO 9000 does not specify high-integrity. The DOD standards provide the maximum requirements and the IEEE SQAP standard provides the minimum requirements. Tailoring is permitted in both cases, but with different meanings.

1.4.1 DOD 2167A, 2168

For the DOD standards, "tailoring" means the deletion of requirements that may not be applicable. However the tailoring does not allow for merging documents or adding documents that may be more appropriate.

1.4.2 IEEE 730

For the IEEE standard, for non-critical software, a subset of requirements may be applied. This standard also allows additional content for an SQA plan. Thus in the IEEE standard, "tailoring" means either addition or deletion of requirements.

1.4.3 ISO 9000

No tailoring is mentioned in ISO 9000.

1.5 Customer/Developer Responsibilities

1.5.1 DOD 2167A, 2168

The DOD standards are intended for use in contractual situations addressing only the developer responsibilities, not the customer responsibilities. DOD (the customer) may be involved in reviews or receive results of the SQA activities.

1.5.2 IEEE 730

The IEEE standard does not mention any specific organization, so the standard could be used internally or could be specified in a contract.

1.5.3 ISO 9000

ISO 9000 is intended to be used in contractual situations and addresses both developer and customer responsibilities. It provides quality system requirements for the developer, including appointment of a management representative responsible for ensuring proper implementation of this standard. ISO 9000-3 is intended to facilitate the application of ISO 9001 to software. An important addition to ISO 9000-3 is a section of requirements for the customer that addresses the customer's requirements and arrangements with the developer. An entire section of ISO 9000-3 identifies customer responsibilities.

In all three standards, requirements on contractors are passed on to their subcontractors.

2. DIFFICULTIES OF IMPLEMENTING SQA STANDARDS

Our preliminary interviews have been primarily with people required to use DOD 2167A and those attempting to use IEEE 730. Problems cited by the interviewees included technical requirements for documentation, and inadequacies of current organizational processes and technology to support the requirements of the standards. The authors' personal experience also suggests that senior management commitment is an issue.

2.1 Documentation Issues

Both DOD 2167A and IEEE 730 specify documentation requirements. A frequent complaint about applying SQA standards was the amount of required documentation. Other complaints took issue with inconsistencies between the documentation requirements and the real world. These users felt that documentation would only add significant value to a product when technology allows the documentation to be produced more quickly and to be maintained consistent with the software.

Because DOD 2167A's allowance for tailoring means eliminating documents or eliminating requirements from them, developers chose to tailor out what they found inappropriate for their contracts and then (justifiably under the tailoring definition) claim conformance with the standard.

2.2 Current Processes and Technology

A set of issues brought up in the interviews dealt with modern technology. Today developers are using various CASE tools that provide forms of documentation but not necessarily in the format-specific requirements of the DOD standards. One developer suggested that the output of CASE tools, rather than the currently required documents, should be acceptable.

2.3 Senior Management Commitment

In the authors' experience, the process of SQA has been initiated at the project level. While tolerable for small projects, the approach seems to collapse as projects increase in size and complexity. The necessity to interact with other project staff for coordinating, performing, and using results of SQA requires a higher level of managerial sponsorship. This sponsorship is necessary to approve and support changes in project schedules and personnel assignments to accommodate the increase in interactions and tasks. The commitment to SQA activities from upper management depends on an understanding that the

additional time and effort early in a project will reduce the need for rework at later stages of the project.

It is interesting to note that this issue was not addressed by the SQA personnel. However it was addressed by the TQM personnel.

3 TOTAL QUALITY MANAGEMENT

The concept of Total Quality Management (TQM) is now well known to many people throughout government and industry. Widely attributed to the pioneering work of Dr. W. Edward Deming and Joseph M. Juran in post-World War II Japan, this concept of "quality first" involves fundamental and large-scale changes in an organization's management and culture. [3]

TQM is a structured organizational effort to implement continuous process improvement in activities throughout that organization. This effort involves: top management commitment, total ongoing participation of management and staff in every aspect of that organization's business, and continuous measurement and improvement of every process throughout that organization.

Deming describes this process by emphasizing 14 points for management to follow in implementing a total quality program in their organizations, see Figure 2. These principles and similar principles advocated by other quality experts are being adopted by many organizations today as they strive to remain competitive in government and private sectors alike.

3.1 Total Quality Management Programs

The TQM philosophy has been formalized in several programs that are focused at different audiences but share common elements. Three major TQM programs will be described: the DOD TQM program, the Malcolm Baldrige National Quality Award (MBNQA), and the ISO 9000.

3.1.1 DOD Total Quality Management

The TQM program developed by the DOD * is designed to change the quality culture of the defense establishment, including its contractors and their principal subcontractors. In this program, TQM is concerned with every managerial,

---

* DOD5000.51-G, Total Quality Management: A Guide for Implementation, February 1989.

1. Create constancy of purpose towards improvement of product and service.
2. Adopt the new philosophy. We can no longer live with commonly accepted levels of delays, mistakes, and defective workmanship.
3. Cease dependence on mass inspection. Require, instead, statistical evidence that quality is built in.
4. End the practice of awarding business on the basis of price tag alone.
5. Find problems. It is management's job to work continually on the system.
6. Institute modern methods of training on the job.
7. Institute modern methods of supervision of production workers. The responsibility of supervisors must be changed from numbers to quality.
8. Drive out fear, so that everyone may work effectively for the company.
9. Break down barriers between departments.
10. Eliminate numerical goals, posters, and slogans for the workforce asking for new levels of productivity without providing methods.
11. Eliminate work standards at prescribed numerical quotas.
12. Remove barriers that stand between workers and their right of pride to workmanship.
13. Institute a vigorous program of education and training.
14. Create a structure in top management that will push every day on the above 13 points.

Figure 2 Deming's 14 Points [3]

design, development, manufacturing, quality, and administrative process that can affect the final outcome of a product. Every functional element in DOD and the defense industry must become aware of their process shortcomings and devise ways to improve these shortcomings.

In regard to product quality, TQM expands the definition from the conventional approach of eliminating defects. Quality first begins with a definition of the correct requirements. When these requirements have been met, primarily

through continuous process improvement, then total customer satisfaction can be achieved. [4]

3.1.2 Malcolm Baldrige National Quality Award

This program was originated by a 1987 Act of Congress and is administered by the National Institute of Standards and Technology, Department of Commerce. [5] This quality program shares many of the attributes of DOD's TQM program but is directed towards the private sector of American business instead of the defense sector. This highly publicized program annually grants the National Quality Award to companies in three categories: manufacturing companies, service companies, and small businesses. Up to two awards may be given in each category each year *. Elements on which the applicants for the MBNQA are evaluated are leadership, information and analysis, strategic quality planning, human resource utilization, quality assurance of products and services, quality results, and customer satisfaction.

The MBNQA, like the DOD TQM program, is characterized by a "continuous improvement" approach. This approach encompasses all operations and processes and includes reducing errors and defects, enhancing value to the customer, improving responsiveness and cycle time performance, and improving efficiency and effectiveness. The continuous improvement approach also emphasizes regular cycles of planning, execution, and evaluation, and using sound quantitative bases for decision making.

Although they address different audiences, both the DOD TQM program and the MBNQA program share some common characteristics. They are focused on customer satisfaction, continuous process improvement, and, most importantly on top management's responsibility to set the quality direction and goals for the company and then follow through with action.

3.1.3 ISO 9000

ISO 9000 addresses quality management systems for hardware, software, process materials and services. Hence, it addresses a limited number of activities, and differs from a system whose purpose is to achieve total quality management. For a TQM system, *every* activity in the organization must be included. [6] ISO

---

* Past winners of the MBNQA are: 1990 - Cadillac, Federal Express, IBM Rochester, Wallace Co., Inc; 1989 - Milliken & Co, Xerox; 1988 - Westinghouse, Motorola, Globe Metallurgical.

9000 emphasizes preparing a quality plan and a quality manual appropriate for the level of quality system required.

3.2 Some Applications of TQM to Software Development

TQM principles, which have been advocated by numerous quality experts, have generally been applied to manufacturing-oriented aspects of organizational environments. Although the machine-intensive, repetitious production lines of manufacturing environments differ greatly from the people-intensive, intellectual life cycle of the software development process, the quality principles of TQM are transferable.

It has been proposed that the general attributes of TQM philosophy are equally applicable and necessary to the systems development process as they are to any other process in an organization. [7] These attributes applied to the systems development process are:

1. Manage systems development as engineering.

2. Manage processing operations as production under statistical control.

3. Concentrate on the motivation and qualifications of every professional, technical, administrative, and managerial employee.

4. Manage the information management unit as a product and service business. Make certain that you not only provide superior products, but also make no less certain that you provide superior service response to your customers.

5. Institute systems for powerful comprehensive management of quality as well as systems for powerful comprehensive management of information.

6. Make quality improvement a central strategy of the information management business. [7]

TQM in the systems development process relies upon many of the same principles that drive TQM in the rest of the organization. Management commitment and responsibility, employee participation, and statistical analysis are all critical components of a TQM plan. Some of the TQM tools which have been suggested as helpful in enhancing software systems quality are process flowcharts, brainstorming, risk management, cause and effect analysis, defects analysis, and

design reviews. Many of these are traditionally used in software verification and validation, a discipline used in performing SQA. [8]

4 PRELIMINARY RESEARCH RESULTS

4.1 Methods - Interviews *

Interviews were conducted with seven individuals at six Federal agencies and companies; all the companies performed contractual work for the government. The interviewees' responsibilities included software development, program management and government consulting. All interviewees were assured of confidentialty.

The interviews were based on an interview guide prepared by two of the team members. The two team members worked together on a preliminary analysis of the interviews in order to identify potentially important issues and hypotheses. Next, one of these two team members coded each interview according to categories that emerged both during the preliminary analysis and during the coding processes. Based on the coding, this team member performed a second analysis.

The discussion below is based on an analysis of this first set of interviews. The interviews served two purposes:

- to pilot test the interview questions
- to identify important issues for further investigation

4.2 Major Issues

Three major issues arose from the preliminary set of interviews:

1. Flexibility and tailoring of SQA standards
2. Customer versus developer in the use of SQA standards
3. Relationship between SQA and TQM organizationally

4.2.1 Flexibility and Tailoring of SQA Standards

The need for flexible standards was a dominant theme in these interviews. Flexibility was tied to the idea of "tailoring", i.e., the ability to use those parts of

---

* The interviews were conducted by one of the authors from American University

the standard that seem most appropriate to their contracts and to delete the remaining parts. Most interviewees wanted to be able to tailor standards. In those organizations that had a choice, the DOD standard was not used in its entirety. According to one interviewee, the DOD standard focussed on "petty and irrelevant" detail, thereby emphasizing format over content. A program manager's comment was:

> "The DOD structure is rigid. In DOD, they tell you more about margins, what kind of print to use."

Several interviewees thought the DOD required documentation was a waste of time for the developer and useless for the customer. As one manager of software development commented:

> "It kills us trying to produce documentation for DOD. It requires as much labor to produce the documentation as to write the software."

According to the interviewees in organizations that must use DOD standards, there has been considerable pressure to whittle away at the standard. Contractors requested that they be allowed to tailor, and the overall effect has been one of reducing the standard's requirements.

Alternatives to the DOD standards were spoken of in more favorable terms. One organization liked a British standard BSI 5750 (authors: identical to ISO 9000) because it fitted well with its emphasis on customer satisfaction. Another interviewee characterized the IEEE standard as a good one because it was simple, lacked detail, and therefore was flexible. However, this interviewee thought that customers did not like the IEEE standard for precisely those reasons. This interviewee thus explicitly raised the issue of a potential conflict between what the developer seeks in standards and what the customer seeks.

4.2.2 Customer versus developer in the use of SQA standards

There was some disagreement among interviewees over what the customer wants. According to one interviewee, customers wanted detailed standards to be adhered to, which is why DOD has the standards it has. A good relationship between the developer and the customer would reduce the need for having everything included in the standards so as to address all of a customer's concerns in software development. However, particularly in government projects, where there may be many bids by contractors, the customer has taken the lead in defining standards, with the ensuing rigidity and detail of the DOD standards.

According to the developers who were interviewed, developers' and customers' concerns differed because the developers did not want to be bound by rigid, detailed standards. However developers see customers as wanting a very detailed standard that addresses all concerns in software development. As one developer put it:

> "The intention (of standards) is to give the customer a warm feeling that everything is in good shape."

However developers did express some concern that customers should be knowledgeable so as not to end up attempting to manage a project using methods with which they have little experience or imposing standards that are, at best, unnecessary.

4.2.3 Relationship between SQA and TQM organizationally

In the companies participating in the initial study, there was little or no relationship between SQA and TQM personnel. In those organizations in which a TQM program was implemented, SQA was seen as supporting and fitting the TQM program. SQA was seen as being part of the TQM philosophy of customer satisfaction and process improvement. As an example, previous quality measures such as the number of errors per lines of code were changed to ones more relevant to customer satisfaction, such as the number of customer requests for enhancements per year.

It is not surprising that there is little overlap between SQA standards activities and TQM. SQA standards, at least DOD standards, are seen by those who must comply with them as being focussed on documentation. In contrast TQM is focussed on customer satisfaction and process improvement. TQM personnel made little mention of software standards. Instead they commented on those aspects of SQA that fit better with the TQM philosophy.

4.3 Future Research Directions

The preliminary set of interviews was used to explore the area of SQA and TQM and to determine potentially important issues. Our initial analysis suggests the following topics for further study:

1. Flexibility and tailoring of SQA standards: can standards be written that are flexible yet rigorous?

2. Customer versus developer in the use of SQA standards: can the two different viewpoints be reconciled so that SQA standards can be used effectively to improve the quality of software?

3. Relationship between SQA and TQM organizationally: what are the barriers to SQA and TQM having interrelationships organizationally?

4. Overlaps between SQA and TQM activities: what are these and can the overlaps be used in some way to improve the organizational structure supporting software quality management?

REFERENCES

1. For DOD standards, order from:

Commander, Space and Naval Warfare Systems Command
ATTN: SPAWAR - 3212
Washington, DC 20363-5100

For ISO standards, order from:

American National Standards Institute (ANSI)
1430 Broadway
New York, New York 10018
(212) 642-4900

For IEEE standards, order from:

The IEEE Computer Society
Order Department
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
1-800-CS-BOOKS or (714) 821-8380

2. Tiratto, Joseph, "Preparing for the EC 1992 in the US through Quality System Registration," IEEE Computer, April, 1991, pp. 70-72.

3. Walton, Mary, "The Deming Management Method", Putnam Publishing, NY, 1986.

4. Perry, William E. "Pursuing Quality Between TQM and NQA," Government Computer News, November 26, 1990, p.50.

5. 1991 Application Guidelines for Malcolm Baldrige National Quality Award, National Institute of Standards and Technology, Administration Building -- Room A537, Gaithersburg, MD 20899.

6. Oakland, John S., Total Quality Management, Butterworth Heinemann Ltd., Oxford, UK, 1989.

7. Schwartz, Herb, "How to Make TQM Happen in Systems Development," Seminar at David D. Lattanze Center for Executive Studies in Information Systems, Loyola College, Baltimore, MD, April 30, 1991.

8. Wallace, Dolores R. and Fujii, Roger U., "Software Verification and Validation: An Overview", IEEE Software, May 1989, pp.10-17.

In The Eye of the Storm

Noelle Evans
Mentor Graphics Corporation
8005 S.W. Boeckman Road
Wilsonville, OR 97070-7777
Arpa Address: nevans@mentorg.com
(503) 685-7000

Noelle Evans is currently a quality engineer at Mentor Graphics Corporation. Prior to joining the DSS project she was a technical writer involved in documenting the next generation software developed at Mentor Graphics. She received her B.A. with honors in history from Portland State University.

Mark Seyler
Mentor Graphics Corporation
8005 S.W. Boeckman Road
Wilsonville, OR 97070-7777
Arpa Address: marks@mentorg.com
(503) 685-7000

Mark Seyler is currently the project leader for the DSS product at Mentor Graphics Corporation. Prior to bringing DSS to market he was involved in research activities aimed at integrating expert systems technology into the EDA design process. The design of DSS is an outgrowth of that research. He received his B.A. in Fine Arts and his M.S. in Computer Science at the University of Illinois.

Abstract

The eye of the storm is a space of relative calm. This paper describes the strategies and tactics used by the Decision Support System (DSS) project team to remain in the eye and bring our product to market. We discuss what we consider to be our best practices, processes, and techniques. Software engineers, quality engineers and project leaders will find this material of interest.

Acknowledgments

We would like to acknowledge the project team responsible for creating DSS. They are John Thienes, Jose DeCastro, Justin Ting, and Sandeep Ajmani.

Thanks also to Bill Stevens for providing us with much needed marketing direction and support at a critical juncture.

A special thanks to Gerry Langeler for believing in the project and for supporting and encouraging us throughout its development.

1. Introduction

The Decision Support System (DSS) is a toolkit that allows users to create custom electronic design monitoring and analysis applications. These applications may be as small as a special purpose calculator or as large as monitoring panels that collect, analyze, and present data.

DSS combines a familiar spreadsheet user interface with a powerful and sophisticated compute engine, a set of interfaces to external data, and an easy to learn visual building-block toolkit.

DSS was developed in conjunction with a larger company-wide effort to revamp our entire software development environment. This effort began in early 1986 with the initiation of a variety of R&D efforts. Mentor Graphics committed to building the development environment using new technology: the C++ object oriented programming language. The development environment consisted of a set of core utilities, a user interface with a programming language, and a design data base. By 1989, the Falcon project, as the new development environment had become known, was concentrating completely on producing a marketable set of products.

Concurrent with the development of the Falcon Framework, we were building applications such as DSS using the Falcon Framework. The DSS project team members were new to one another and to the C++ object oriented technology. Developing a new application on top of a changing software development environment using new technology by a newly formed product team made for a challenging and dynamic environment. The additional pressure to bring products to market as quickly as possible made us feel as if we were in the midst of a great storm.

Although the Falcon project and the release was a major undertaking for Mentor Graphics as a company, we do not believe the pressures and requirements surrounding the DSS project were unique. All project teams face challenges and expectations that have the potential of diverting resources which may ultimately jeopardize the quality of the end product. We believe that amidst the "storm" that accompanies new software product development there exists a space of relative calm; the eye of the storm. This paper describes the strategies used by the DSS project team to remain in the eye and bring our product to market.

2. Project Strategy

2.1 Team Organization

Organizing the team to work efficiently and confidently was a critical element in the success of DSS. Our organization was the result of trying to minimize the frequency and maximize the quality of our communications while still disseminating the information each team member required.

The initial engineering team that brought DSS to prototype was a single R&D engineer. Rather than handing off the prototype to an implementation group, the initial designer remained with the project. As the project moved from the R&D phase to the development phase, the team, now three engineers, made a conscious decision to keep the overall size of the project team small. Other team members were cycled in only as needed. We added a marketing resource first, then a technical writer, and finally a quality assurance engineer. We found timing to be critical. For example, our quality assurance engineer was added late in the development phase, stretching our resources to put tests in place and increase our test coverage to meet our corporate targets as we neared our alpha release date. These difficulties would have been minimized if the QA engineer had joined the project earlier and been able to spend more time creating the test suite.

During this early research and development period, the division was trying to improve team organization. Traditionally, quality assurance engineers were physically located with development engineers, but other team members were not. In 1988 and 1989, technical writers moved in with the projects. In 1990, the marketing representatives and customer support engineers also located with their projects. This physical proximity proved to be an important contributor to overall efficiency of communications.

We organized the project team into empowered specialists. We clearly defined ownership and responsibilities and distributed the decision making authority to the team members. We established an environment where making some mistakes or bad decisions were acceptable.

This structure was a challenge to implement. The first challenge came when the project leader had to share authority. This meant that the project lead had to give up knowing everything about the project in detail. The project lead role emphasized information brokering and supplying team members with the information necessary to accomplish their tasks. This information ranged from company priorities to project priorities. The second challenge was that project members had to accept responsibility. Team members had to assume the decision making for their particular parts of the project and be accountable. We also had to provide the project lead with the appropriate information for our clients.

We initially introduced very few processes. These processes were informal contracts describing some operating procedure, often between just two people. We only added new processes after:

- finding a difficulty either within or outside the group or in the group's interactions with others
- identifying and clearly understanding the problem
- determining that a new process was the only way of resolving the problem

The goal was to maintain individual productivity by avoiding unnecessary bureaucracy.

We increased our efficiency by avoiding committee decision making situations. Instead, we encouraged either unilateral decision making or decisions made by pairs of people. Oversight and review occurred in small groups through informal and well defined processes. For example, designs were regularly reviewed by the design engineer and the technical writer. The engineer explained the design and presented the design notes to the writer. If the writer objected that the explanation was unclear or too complicated, the engineer either clarified the design notes or re-did the design.

This team structure improved motivation, efficiency, and communications. Pride of ownership and knowing that decisions made by individuals could impact the success of the entire project motivated the team to strive to do their best. Because each team member knew the responsibilities of the other team members, internal communications became more efficient. Questions and concerns were directed to the appropriate person the first time rather than being shuffled from member to member. External inquiries were adversely impacted only to the extent that not everyone outside of the project knew how responsibilities were divided. However, each team member took the responsibility of directing the inquiry to the appropriate person.

The strong project identity and the confidence that team members gained from it helped the team to weather the storm. Changes that occurred in the surrounding development environment had less impact and rarely disrupted the project.

2.2 Prototyping

We developed a working version of DSS quite early in our development process and maintained it until the product shipped to customers. We found the costs of maintaining this working version to be fairly high, but the long term benefits outweighed the costs. Because DSS and the Falcon Framework were developed concurrently, we had to implement major pieces of the system that would later be replaced by the Falcon

Framework source code. On the one hand we developed disposable code, but on the other hand that very code insulated the project from major functionality changes made to the Falcon Framework. We had the freedom to gradually migrate to the new source code or choose low risk periods in the development cycle. We used the prototype both internally and externally as a means of communicating our product vision. It also let us identify and correct usability problems early in the development cycle.

Along with the prototype, we focused on end-user documentation rather than writing a functional specification. We maintained a usable version of the reference manual and quick reference guide throughout the life of the project. We set our documentation priorities by estimating document longevity and audience size. Documents with an expected long lifetime and large audience got top priority.

Having a working prototype and user documentation gave us continuous feedback from internal and external customers. We knew that if we were to be successful, we needed to listen and respond to our customers. Knowing our customers needs gave the project team a common, indisputable goal that we were doing the correct tasks.

2.3 Planning

Planning was an essential component for us to remain in the eye of the storm. Knowing what our priorities were and thus what needed to be done and how long it would take to do it provided us with all the protection we needed against the storm.

We identified tasks and assessed their cost several times within a release cycle. A typical tasking cycle, where we assessed the time it takes to do a task, took us about 30 to 45 minutes. The result was a detailed task list which was available at all times to team members. Regularly identifying and assessing tasks gave us a chance to revisit our priorities and check our progress.

Identifying tasks was not always easy. The most difficult aspect in identifying tasks was defining the granularity of a task. Should a job that took an hour be considered a task? Should jobs that took at least a half day be considered a task? In the end, team members defined their own task granularity. Most of the team considered a task to be a job that took at least a day. Jobs that took less than a day were accounted for in the amount of overhead included in the schedule.

The responsible team member identified and assessed the cost of their own tasks. Team members collaborated on group tasks. Our primary estimation method was to compare similar tasks. Initially, we were poor at estimating the cost of a task. We carefully compared our estimated cost of a task to the actual cost to complete the task; this helped us make better cost assessments the next time. Our goal was to get good at estimating the

cost of a task, not to keep score on how well we did.

We became good at estimating known tasks but found that unexpected tasks were causing us to miss our targets. Unexpected tasks were primarily tasks we failed to identify that had to be completed or tasks that were added after our planning cycle such as new customer requirements. We measured the impact on our schedule of these "unexpected tasks" over a period of several tasking cycles and now add a multiplier. For example, approximately 10% of our engineering time was spent completing unexpected tasks, so we multiplied the total time to complete all our tasks by 10% and added this figure to the total. We continue to monitor the amount of time spent on "unexpected tasks" and adjust the multiplier accordingly. Another variable we monitor and add in as a multiplier is our cost to support clients using DSS.

When we found dependencies between tasks, we attempted to assign them to one person, even if doing so meant some additional training or reorganization of the source code. Our purpose was to reduce the dependencies and communications overhead within the project and increase our productivity.

This sort of planning was a critical asset to the project. We gained experience in producing accurate time estimates. We now have the reputation for providing and keeping detailed and accurate schedules.

2.4 Pacing

We knew from the early phases of development that it would take several years to get the product to market. We were concerned that the team remain intact for the duration. We developed a work pace and style designed to encourage sustained productivity. Our objective was to work smarter, not harder. We planned and scheduled our tasks very carefully. We focused on the tasks scheduled and guarded our concentration by capturing issues of the day but deferring our responses to them.

Capturing issues but deferring our responses was perhaps the most important action we took to pace ourselves and complete the project. It kept us focused on our priorities. We often found that, in retrospect, the issues of the day became a much lower priority than they originally appeared. Occasionally, they went away completely.

2.5 Software Tools and Techniques

For software version management, we used a tool available from HP/APOLLO called "DSEE". We assigned a single engineer to master this tool, maintain it, and provide training for other engineers. Our software release process was also managed by a single engineer who had complete responsibility over the release process.

A technique we called "usage analysis" proved very valuable during software development. It involved a real-time search of various text databases in order to generate cross reference information. This technique required only simple pattern matching technology and access to source text over a network. With the AEGIS "fpat" pattern matching tool, similar to the UNIX "grep", we were able to write shell scripts to quickly find instances of function or class usage in unfamiliar source code. This gave us the most reliable and up-to-date information. For this project, we developed over 50 scripts to search everything from problem report data bases to source code repositories and system libraries.

To understand some large subsystems, this simple technique proved to be more useful than analyzing more conventional forms of support documentation. It let us focus on the problem and simultaneously allowed us to gradually understand a design. It gave us a client's picture of the design and let us determine how well client's requirements were being met. Many times we were able to simplify our design based on this information.

Much of our project information was stored in simple text files. We accessed this information via a project notebook. The project notebook consisted of another set of text files that contained mostly file references. A simple key definition let us point and click on one of these pathnames to bring up its view. In use, the system had a "hypertext" like quality. All of these files were stored in a single directory. At the time of this writing, that directory contains over 2000 files, all related to some aspect of the DSS project.

Although not sophisticated, this simple text database served our needs well. Information access was very fast, due largely to the presence of the reference notebooks. Many team members had similar, personal notebooks with links to the master text files.

We used assert statements in the source code to improve our software quality. Assertions act like correctness monitors. They are best captured as code is designed, when assumptions are fresh in your mind. Because they do not change the behavior of the code, they can be added at any time.

We found that errors caught by assertions are cheaper to fix than other types of errors. We suspect that this is because they tend to uncover problems at the point where the problem occurs. Our source code contains over 900 assert statements, or about 2% of the total source. 17% of all the errors found in DSS were found by assert statements.

3. Test & Measurement

3.1 Test Strategy

We faced the challenge of assuring our customers that DSS had the functionality, usability, reliability, and performance they expected. We used a number of strategies to reach our goal.

Most of the tests developed for DSS were at the system level. System-level testing divided into the following areas: user interactions, DSS functions, ASCII and Code Level interfaces, application development, and help testing. In addition, we ran performance tests, configuration tests and integration tests.

Wherever possible, tests were automated using a variety of techniques. The primary technique used the DSS ASCII metafile. Because the metafile had a predictable format, we could generate the DSS test metafiles using internal tools that both documented and generated test cases. This allowed us to efficiently add to the existing test suite and still maintain existing test cases. DSS metafiles execute when loaded or when specific function calls are made. Therefore running the test suite essentially consisted of loading the test metafiles into DSS and evaluating the results. This test methodology covered 90% of DSS functionality. The remaining 10% divided between the graphical interface and the userware. The graphical interface tests were visual inspection tests. Because DSS was built on top of the Falcon Framework, the UIMS project team tested much of the user interface. The userware was tested by replaying physical transcripts. A physical transcript is a file of low level screen coordinates and key actions. Initially, physical transcripts were generated manually by logging the actions and saving them to a file. An internal tool was developed that allowed us to specify in an ASCII file the actions in a high level language. The file was then "compiled" into the screen coordinates and key actions. We ran the test by replaying the file in DSS.

We implemented new or changed functionality that was high risk to the product as early as possible in the development cycle. The changes were released to the project team as soon as the development engineers believed it was stable so that everyone had the opportunity to use the new or changed functionality and in general lived with the new release. There often were a number of these "mini" releases depending on how much of the source code needed to change. Although we created formal test cases for all new and changed functionality, this general usage testing by the project team often found interactive and multi-task problems that were difficult to find as part of the formal testing.

We used test coverage analysis to prioritize the test case development. We used an internal tool to compile the source code. The binary produced by the tool was instrumented so that it counted the number of times a test or test suite executed a

statement. Once the tests ran on the instrumented binary, we generated a report that listed the percentage of the code exercised on either a per module or per function basis. This information told us which areas of the code we were not testing or were only lightly covering. QA and development engineers worked closely to identify critical areas of the source code to test and the specific test cases that would exercise the code.

In four months, our test coverage went from 0% to 70%. We also had an automated acceptance test suite and an automated regression test suite that we could use to test phased releases. In nine months, at the Code Freeze milestone, we had 84% test coverage and all major areas of functionality tested. This exceeded our highest goal, which was 80%. During this period, we supported one Alpha release and three Beta releases to customers who were willing to evaluate our software. In addition, we supported a number of customer and conference demonstrations.

3.2 Quality Measurements

We measured our quality based on four attributes:

- Functionality
- Usability
- Reliability
- Performance

In addition to these attributes, we established specific Release criteria for releasing the product.

3.2.1 Functionality

Functionality was measured in two ways:

- Test coverage
- User feedback

Our test coverage, as described previously, was measured by an internal tool applied at compile time. The tool instrumented each executable statement so that when we ran our test suites on the instrumented binary code data was collected on the statements exercised. We generated reports using the collected data which gave an overall coverage figure as well as coverage data on specific modules and functions.

Our goal for DSS was to reach 80% coverage by the release date. We exceeded our goal and reached 84% test coverage. We accomplished this by regularly running our tests on an instrumented version of DSS. In a six month period we ran our tests on an instrumented version of DSS four times. This allowed us to evaluate where our coverage was weak and create specific tests to exercise that aspect of the code.

In addition to test coverage analysis, we asked users to rate product functionality. Internal users and beta customers, approximately 6 sites, were asked to rate the degree to which the product possessed the necessary and sufficient functions. We used a scale from 1 to 5, where 1 was very unsatisfied, and 5 was very satisfied. DSS had an average score of 4.6 points.

3.2.2 Usability

Usability was measured in two ways:

- OSF/Motif compliance
- User feedback

Motif is a user interface standard established by the Open Software Foundation (OSF), of which Mentor Graphics is a member. In addition to the Motif standard, Mentor Graphics extended the interface to include specific items our customers were accustomed to using so as to provide a common user interface. Much of the Motif standard was implemented in the Falcon Framework that DSS was built upon, but DSS controlled some areas. We completed the Motif/MGC extended checklist showing that DSS was compliant with the standard.

We also asked our users to rate our usability. They were asked to rate the amount of effort required to understand, learn, and use the product. They used a scale from 1 to 5 where 1 was very unsatisfied and 5 was very satisfied. DSS had an average score of 4.1 points.

3.2.3 Reliability

Reliability was measured in the following ways:

- Declining defect discovery rate
- Open defect count
- Weighted defect count (WDC)
- User feedback

Defect discovery rate is the rate at which problem reports are filed against the product. Our goal was that the defect discovery rate would decline for 5 weeks prior to the Code Freeze milestone. Our defect discovery rate declined for 4 weeks prior to the project code freeze date.

The open defect count is the number of open, and therefore unaddressed, problem reports. We met our goal to have the open defect count not include any severity 1 or 2 reports. Severity was determined using the following standards: severity 1 reports caused a crash under normal operating conditions, resulted in corrupted data, or produced misleading results and there wasn't an alternative that could be used to obtain the same functionality; severity 2 reports were the same as a 1 but an alternative method was available; severity 3 reports were a significant deviation from the product documentation; severity 4 reports were a minor deviation from the product documentation, inconsistent, or inconvenient to use.

The weighted defect count (WDC) is a weighted count of all the open problem reports against the product. Severity 1 reports were valued at 6 and low severity reports were valued at 1. Our goal was 27 weighted points or less at code freeze. This value reflects 6 weighted points per 10,000 lines of non-commented source lines of code. Our actual WDC count was 43. Although we did not actually make our target, we went back to all our clients and asked that they inform us of any release critical reports, regardless of the severity. Our clients assured us that the remaining open reports were not release critical.

In addition to asking our clients about release critical reports, we asked them to rate the reliability of DSS. They were asked to rate the capability of the product to maintain its level of behavior under stated conditions for a stated period of time. They used a scale from 1 to 5 where 1 was very unsatisfied and 5 was very satisfied. DSS had an average score of 4.3 points.

3.2.4 Performance

Performance was measured with user feedback. Users were asked to rate the capability of the product to perform the specified functions under stated or implied conditions within appropriate time frames, using appropriate amounts of resources. They used a scale from 1 to 5 where 1 was very unsatisfied and 5 was very satisfied. DSS had an average score of 3.2 points.

Because DSS is a new product, and no other comparable product is on the market, we did not make any assumptions about performance, other than what our users told us. We are in the process of establishing benchmarks that we will use in subsequent releases to measure our performance. We will continue to ask our customers where they find the performance unacceptable so that we can concentrate our energies in the most important areas.

3.2.5 Release Criteria

In addition to the four attributes used to determine whether DSS was ready to ship, described previously, we also used some internal requirements to monitor our progress:

- Phased Release criteria
- Evaluation Start (ES) criteria
- Code Freeze (CF) criteria
- Transfer to Release Team (TRT) criteria

The last three criteria correspond to milestones in the Mentor Graphics Product Life Cycle (PLC). The PLC is a set of milestones, with corresponding check lists, that track the progress of a product from the earliest phase, developing requirements, through the complete life of a product to the final phase, discontinuing the product.

The Phased Release criteria were that the QA acceptance tests passed and all fixes to client critical problem reports were verified. The project team also answered the question "Was this version of DSS better than the last?" If we could not answer a unanimous yes, then we re-evaluated the problems that caused us to respond negatively.

The Evaluation Start criteria were that the phased release criteria were met, the PLC ES milestone checklist had been completed, and all functional objectives were met. In addition, all high severity problem reports had to be resolved and our test coverage had to be 70% or greater.

The Code Freeze criteria were that all the ES criteria were met. All the metrics for the attributes that were previously described had to be met and the PLC CF milestone checklist had to be completed.

The Transfer to Release Team criteria were that the CF criteria were met and that the PLC TRT milestone checklist had been completed.

4. Summary

The eye of the storm, we believe, can be found. Central to finding the eye appears to be a focus on communications issues:

- Increasing communications with potential customers

This allows your design team to stay focused on tasks that satisfy the customer. Early prototypes can help communicate your product as well as provide a forum for continually refining that vision.

- Co-locating team members

  The close proximity of the team facilitates the flow of information. However, controlling communications flow and reducing or eliminating unnecessary communications is important for increasing overall effectiveness of your team.

- Deferring the "problem of the day"

  This strategy can help you stay focused on priorities.

- Providing accurate task estimates

  This allows others in the organization to make and execute plans without continually tracking your progress.

- Empowering individuals

  Sharing responsibility and authority reduces the need for communications.

We have been asked many times to what extent we believe these techniques are applicable to larger groups. Certainly, considerable productivity gains can be had by empowering individual workers; and this empowerment can occur regardless of group size. The skills and techniques in planning and prioritizing are also achievable on an individual level. But many of the communications techniques described in this paper do not appear to scale up to larger groups. In larger groups information flow is complicated by organizational, geographic and temporal barriers. Organizations can be fine-tuned by carefully examining and re-engineering their internal processes. And we are all familiar with the role that electronic mail and video conferencing technologies play in bringing our organizations together. But increasingly one hears that although the information is there, it is not collected and delivered where and when it is needed. People can collect and deliver this information, but people are already too busy. Computer-based tools are needed that can take an active role, rather than a passive role, in the organization. When information critical to the organization changes or when data needs to be monitored on a regular basis, these tools would provide the information without the need for human intervention. Human resources can then be applied to decision making rather than to information monitoring tasks. Although tools of this sort are just now being developed, we think they will play a key role in achieving the next major level of productivity gain.

Experiences with Defect Analysis

Brian K. Casey and Jan L. Sun

ABSTRACT

This paper describes the techniques and tools that have been developed and implemented at Bellcore for the analysis of software defects in software products developed and licensed by Bellcore. The objective of software defect analysis is to identify potential process changes that will enable the software developer to make the greatest improvements in software quality and productivity. Measures of expected quality and productivity improvements are presented from projects that have performed software defect analysis. Practical issues such as resource requirements, and technical and management factors for successful implementation of defect analysis are addressed. Overall conclusions based on Bellcore's experience with performing defect analysis on several projects are provided.

The method used to perform software Defect Analysis requires the analyst to answer several questions relating to the origin of the software defect, such as, where in the software life cycle the defect was introduced, how the defect could have been prevented, and how the defect could have been detected and corrected earlier in the software life cycle. This information is collected and maintained in a database. An analysis of the database can be performed to:

1) Identify the most frequently occurring and troublesome software defects.

2) Determine where in the life cycle most defects occur.

3) Determine the most frequent root causes of defects.

4) Determine the effectiveness of different defect prevention and detection measures.

5) Determine the expected productivity and quality gains of recommended process changes.

Database analysis reports are available in the form of text and bar charts and are reported in a way suitable for management to use to make more effective decisions to allocate resources and time for process improvements.

I. Quality Improvement Through Process Improvement

A. The Costs of Poor Software Quality

1. Costs to the developer

Software costs have been growing at a rate of 12 percent per year since 1980 [1]. Most of this cost growth is due to increased demand for software products with more features and versatility, yet nearly all users expect software products to be more reliable and easier to use with each new release. User demands for increased functionality and improvements in versatility and ease of use result in software which increases in size and complexity with each successive release. Figure 1. illustrates this significant challenge to the software development organization that wishes to improve the quality and reliability of its products. If the software industry is to successfully meet this challenge, it must, at the same time, improve the quality of its software products and the productivity of its development processes.

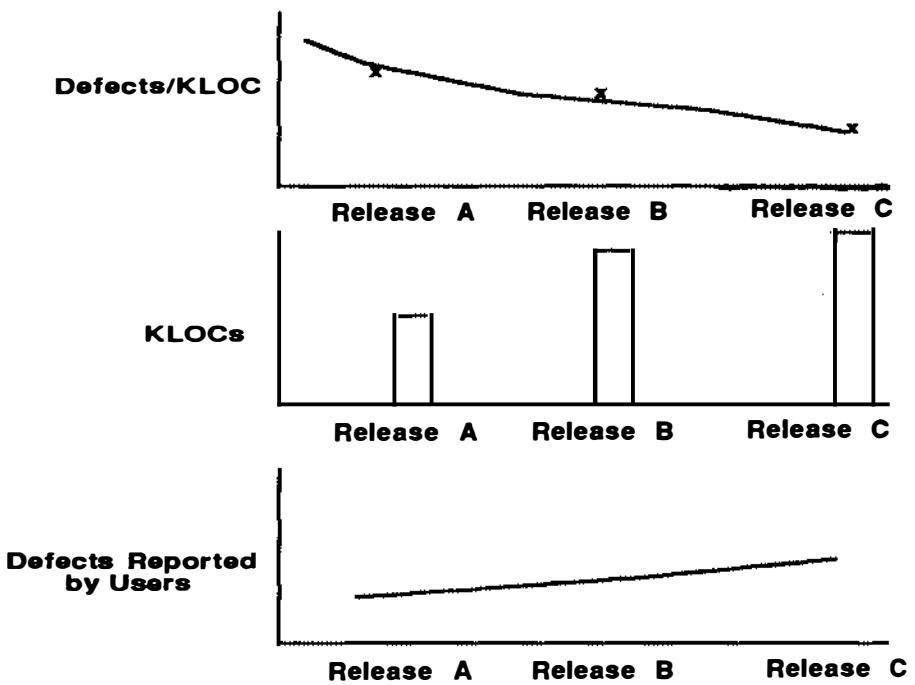


Figure 1. The software quality and productivity challenge

Software quality and productivity experts agree that reducing software rework (i.e., fixing defects and retesting software) will produce the greatest gains in software quality and productivity [1]. Studies have shown that

most companies spend about 20 percent of their time fixing and testing software field defects reported during operation [2]. Cost data collected from many different projects by Boehm has shown that the cost of fixing a field defect is 20 to 1000 times more than fixing defects found during development [3]. Costs incurred by the customer were not included in Boehm's study. Preventing software defects or finding and correcting them in the earliest stages of the software development life cycle will provide significant cost savings while simultaneously improving software quality.

2. Costs to the customer

Bellcore has conducted several internal studies on the costs of poor software quality and has found that software field defects almost always are much more costly to the customer than the developer. The customer must expend resources verifying the software defect, reporting it to the developer, conducting "work around" procedures, and installing and verifying the fix. In addition to these costs of "living with" software defects, software defects in critical portions of a system (e.g., telephone networks, airline flight reservation systems, and electronic funds transfer systems) can result in millions of dollars of lost revenue in a short time. Costs to the developer can easily exceed the cost of fixing the defect if the developer is found liable for the customer's losses. Since there are usually multiple customer sites using the same software, these costs may be incurred many times for a single defect.

B. Improving Software Quality Through "Focused" Process Improvements

Traditional techniques for improving software quality and reliability have relied on removing software defects in the later stages of the development life cycle. While defect removal activities, such as system level testing, often provide immediate and significant improvements in software quality, they do little to provide quality improvements that last throughout the life of the product. Each effort to improve software quality with each successive release must be started anew with increasing levels of testing required to achieve continuous improvement in product quality and reliability. Testing by itself removes only the defect and not its cause. With only test and inspection methods in place, defects are removed one at a time, and the same kinds of defects are often discovered in later releases and in different parts of the product.

Since the late 1980s, there has been renewed interest in the use of process assessments as a quality improvement tool [4]. The process assessment focuses on examining the project's software life cycle methods, procedures, and tools. During a process assessment, interviews with project personnel and reviews of relevant documentation are conducted to determine the level of compliance. The audit findings are compared against an objective

standard for software quality programs (e.g., Bellcore's software quality program standard, TR-TSY-00179 [5]), and a determination is made as to whether the project's quality program meets the standard. While quality improvements based on process assessment findings are longer lasting than those based heavily on testing, the assessment itself may not determine what process changes will produce the greatest improvements in quality.

Defect Analysis is a "hybrid" method that determines what parts of the process are producing the greatest number of defects and what combinations of process changes could prevent the greatest number of defects from being created or will find and fix defects while the software is in its earliest stages of development. Quality Improvement that can achieve these goals can produce the greatest improvements in quality at the lowest cost. Since quality improvements are a result of process improvements, the benefits are long lasting and incremental improvements to product quality can be made economically. Defect prevention techniques that share many similarities with Defect Analysis have been pioneered by IBM and have been successfully used on large software projects in the commercial world [5,6,7,8, 9].

II. Defect Analysis

A."Defect Analysis Paradigm"

The model of a feedback control system shown in Figure 2 best describes the approach Bellcore has taken for quality improvement through process improvement. The control system paradigm requires that critical product attributes that are under control be periodically sampled and analyzed. The results of the analysis are translated into adjustments to the process that will produce the desired change to maintain control. The process continues to be measured to determine if the process remains under control.

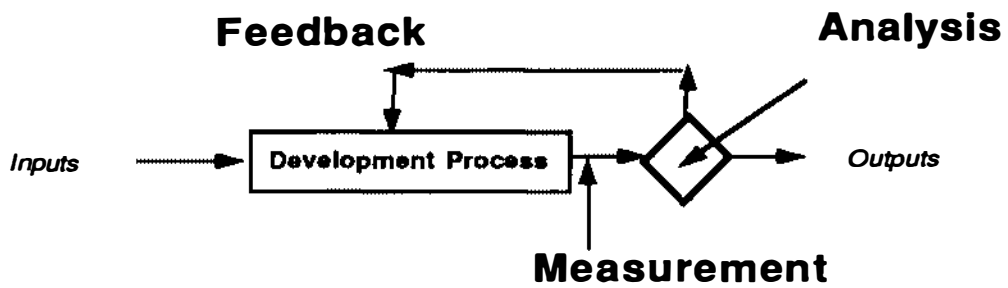


Figure 2. Feedback Control System Model

If we are to implement quality improvements using this paradigm for software quality improvement, the following development practices should be established:

1) software development activities should be "formalized" as a process
2) the defined software development process should be adhered to, and
3) the outputs from the software development process should be measured.

Since improving Software Quality means reducing the number of defects that are produced during development, it seems logical that defects must be sampled (assuming you have so many defects that sampling is needed) and analyzed to determine what process adjustments should be made to improve quality. The analysis of individual defects is based upon finding the defect's "genesis in the process" and answering the following questions:

1) where in the process was the defect created and what caused the defect to occur in the first place (the defect's root cause),
2) how could the defect have been caught at an earlier stage of the software development life cycle (i.e., early detection measures), and finally,
3) what could have been done to have prevented this defect from occurring (i.e., prevention measures).

B. Implementing Defect Analysis

1. Activities

Defect Analysis consists of four iterative steps: planning, analyzing individual defects, analyzing the group of defects and, effecting change. These activities and the sequence in which they are performed is depicted in Figure 3.

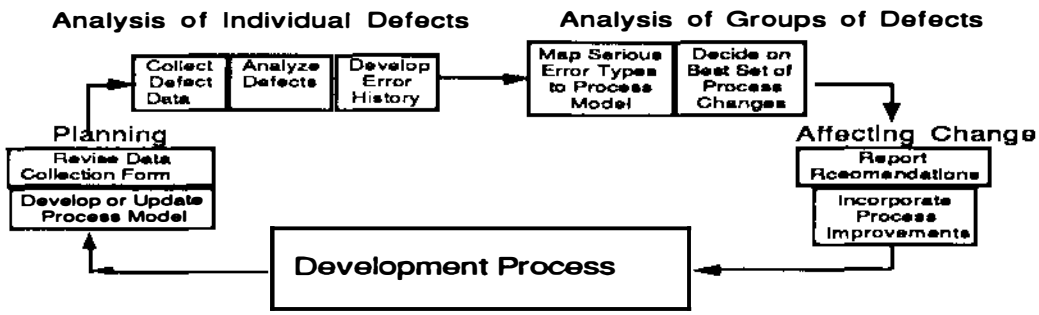


Figure 3. Defect Analysis Model

The **planning** step consists of developing or updating an existing life cycle process model and determining what kinds of prevention and detection measures should be considered as candidate process changes. Process modeling consists of breaking down the software development process into discrete life cycle activities. Each life cycle activity is decomposed into:

1) entry criteria (i.e., required inputs needed before the activity may begin)
2) process steps (i.e., a sequential description of essential activities and products that are produced),
3) exit criteria (i.e., criteria that must be met for products to be considered to have met quality objectives).

Defect Root Causes are related to a failure of a particular life cycle activity to perform as required. Figure 4 shows how candidate prevention and detection measures are created from entry criteria, process steps and exit criteria. The possible Root Causes, Prevention Measures and Detection Measures that are identified during this analysis are used in the defect analysis checklist form (a more detailed description will be provided in the discussion on Defect Analysis tools in this paper).

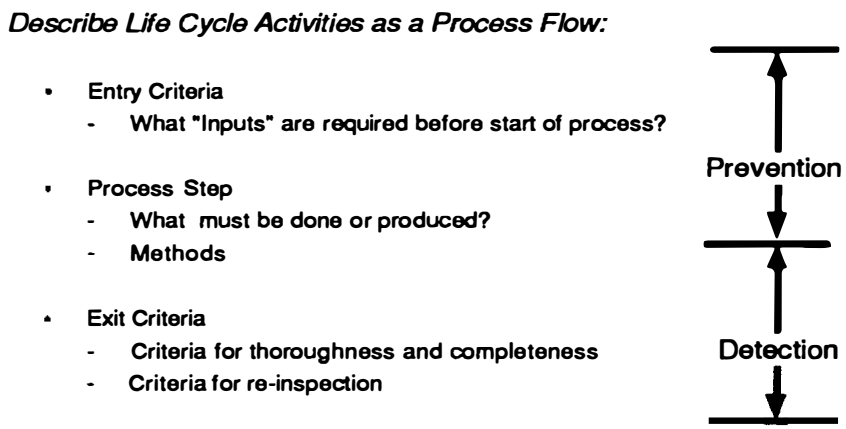


Figure 4. Derivation of Prevention and Detection Measures

Planning also involves resolving administrative issues and performing the groundwork for defect analysis. Decisions are made on administrative issues such as: the kind of defects to be analyzed (field faults, system test faults, etc.); the total number of defects to be analyzed as a group; the frequency of team meetings (once a week, once every three weeks, etc.) Groundwork to be laid is activities designed to help to ensure that defect analysis is worthwhile for the project and to help to ensure that all team members have a common understanding of the process. Depending on whether the project has a consistent and documented process, the team

would either review the process or document the process and use that process to make revisions to the form they will use to analyze defects.

The **analysis of individual defects** begins with the discovery of a defect by the customer or during acceptance testing. An overview of the major activities and players in the defect analysis process is shown in Figure 5. A description of the defect is recorded and sent to the development organization for resolution. Actual resolution of the defect is performed by a product expert (often the individual responsible for that particular portion of the software product). The product expert performs the initial analysis and records his/her analysis on a written data collection form. The form and any necessary backup materials are brought into the team meeting for discussion. The team and the product expert discuss the defect and agree on its root cause. Although the main idea of this activity is to determine the root cause, the analysis also includes measures or techniques for the prevention and detection of the defect. This is because discussion of possible prevention and detection measures often leads to identification of the real root cause. The analysis results are recorded on the form that is used to input the data into a database.

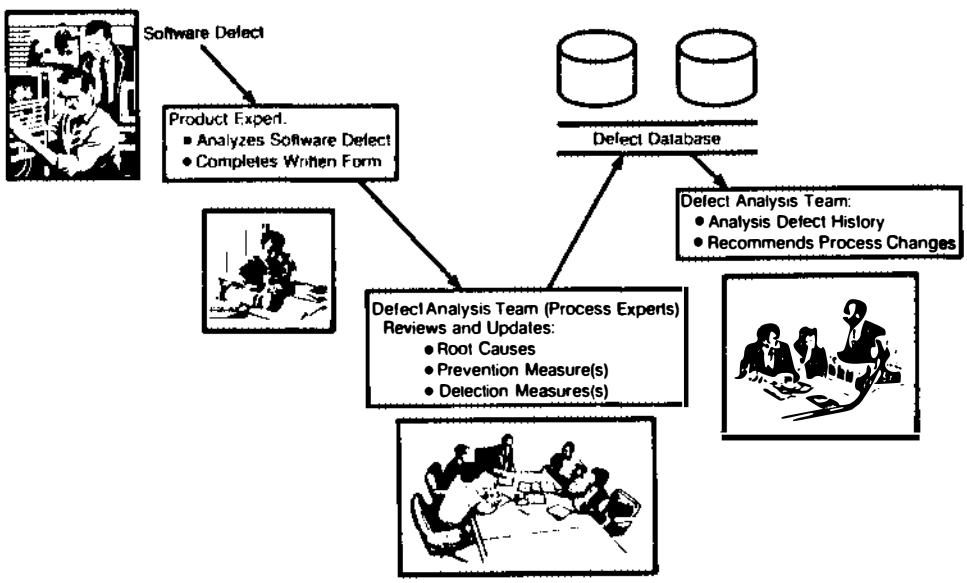


Figure 5. Description of Defect Analysis activities and participants

Analysis of groups of defects is done by the Defect Analysis Team analyzing defect history data and performing analysis using the defect database. The team uses the database to determine specific areas that should be focused on, and to identify any defects that require reanalysis. Most often, a defect is reanalyzed when its root cause is commonly occurring, but an unusual set of prevention and detection measures were recommended. When all of the defects have been analyzed and entered into the database, the team selects a set of the most effective process changes (prevention and detection measures.) Choosing process changes could begin by identifying the

phase(s) with the most defect root causes then analyzing the prevention and detection measures in the phase(s) to decide the best set, or the team could identify the best overall combination of prevention and detection measures across all life cycle phases. The set of process changes should be selected based on the number of defects that would be prevented or detected earlier in the life cycle and on the cost, effectiveness, and ease of implementing the process change.

In order to **affect change** to the process, after all the recommended process changes have been identified, the cost savings to the company is calculated. The results, including the cost/benefit analysis, are documented in a report to management so that the recommended process changes can be carried out. When the changes are implemented the entire process is repeated.

Defect analysis from planning through reporting, when it is necessary to document the development process, takes approximately 25 three-hour meetings to complete. Factors that affect the total number of meetings include: the number of defects analyzed, the amount of groundwork done, the length of discussion allowed for each defect, and the number of defects analyzed per meeting.

2. Participants

The process of analyzing a defect to find its root cause is a team effort. Although the Defect Analysis Team (DAT) consists of "process experts," "product experts" also participate in these discussions. Through discussions among process experts and product experts, a more thorough understanding is developed and the conclusion is more likely to be the real root cause. Process experts would be those who are very experienced on the process that is followed for the various phases of development. For example, a project leader of the requirements team probably would be an effective "process expert" of the requirements phase. A product expert would be someone who is well versed on the "product." For example, the person who wrote the requirements that has an error would be an effective "product expert".

The final determination of the defect's root cause is reached through team consensus; hence, the larger the team size, the longer it has taken to reach consensus. Therefore, the right balance must be achieved between having the right people on the team and the right number of people on the team. Often, the Defect Analysis Team consists of "process experts" to cover the entire life cycle, which suggests that some teams can cover the entire life cycle with three process experts, and some would require six or seven. The "product experts" would only join the meetings or discussions if their particular product is the one being discussed. Once the team begins to

analyze the group of defects, only the process experts are required to participate.

C. Defect Analysis Tools

Tools are necessary to streamline the defect analysis process and to help to ensure that Defect Analysis activities are carried out in a complete and systematic way. The tools that have been useful are:

1) a customized checklist form to aid the selection of root cause, prevention and detection measures,
2) a database analysis tool to store defect data and automate the analysis process, and
3) a report template as a guideline for reporting the results from the analysis.

The **checklist form** provides a method for selecting possible root causes, prevention, and detection measures when performing defect analysis. The form helps to ensure that a consistent terminology and all frequently used root causes, prevention and detection measures are considered when analyzing a defect. Consistent terminology reduces the burden of trying to decipher the analysis and recommendations of different people who have analyzed defects. The checklist form has other advantages. For example, putting the Defect Analysis results into the database becomes a simple job of data entry since the database can be organized as menu selections. Analysis of the data is also greatly simplified. A sample of checklist of selections for root cause, prevention measures and detection measures is provided in Figure 6.

Prevention Measures **Detection Measures** **Root Causes**

User Needs Definition (UND):
1. Define/enhance and document UND procedures
2. Develop guidelines for specific UND procedure
3. Use UND document template
4. Require or enhance formal UND document
5. Establish formal procedures for UND changes
6. Method to estimate/communicate UND impacts
7. Include UND task in detailed schedules
8. Client TAG or Focus Group UND involvement
9. Use SME as consultant during UND
10. Obtain UND input from key personnel
11. Analysis of Client operation or environment
12. Require UND feasibility study
13. Provide PROJECT or UND training/awareness
14. Develop UND "most probable errors" checklist
15. Other (UND):

User Needs Definition (UND):
1. Define and document UND review procedures
2. Conduct external UND review with Client
3. Conduct formal internal UND presentation
4. Ensure key personnel attend UND review
5. " Most probable errors" checklist at UND review
6. Establish formal UND exit criteria and signoff

Functional Requirements Specification (FR
7. Define and document FRS review procedures
8. Training/awareness on FRS review procedures
9. Enhance external FRS review with Client
10. Enhance formal internal FRS review
11. Require SME at FRS review
12. Ensure key personnel attend FRS review
13. " Most probable errors" checklist at FRS review
14. Establish formal FRS exit criteria and signoff
15. Other (UND & FRS):

User Needs Definition (UND):
1. UND procedure not established
2. UND procedure cut short by schedule
3. UND procedure not followed or enforced
4. UND procedure not adequate
5. UND input not solicited or received
6. UND document not produced or incomplete
7. Client need or operation misunderstood
8. Insufficient or late feedback from Clients
9. Feasibility study not performed
10. Expedited development decision was incorrect
11. Resource limitations not considered
12. Lack of PROJECT or UND expertise
13. Timeliness of Client Support
14.
15. Other (UND):

Figure 6. Prevention Measures, Detection Measures and Root Causes

Bellcore has used a **database tool** for performing defect analysis on its own products for over two years. Since each defect in the database often has more than one prevention measure and detection measure identified with it (perhaps as many as six candidate process changes per defect), finding the most effective process changes cannot be done by simply counting the number of times they were selected and identifying prevention and detection measures that were most often recommended during the course of the Defect Analysis. The database tool keeps track of the defects that are eliminated by a particular combination of process changes. The tool presently can analyze different combinations of prevention or detection measures. The database is used to eliminate the "overlap" created when a defect is eliminated by more than one prevention and detection measure and show the true effect of carrying out these process changes. These calculations can be quite tedious when more than a couple process changes and defects are analyzed. Figure 7. illustrates the "overlap" situation that must be accounted for in the analysis process. Allowing a defect to be eliminated by more than one process change means that defects must be carefully accounted for when analyzing process changes.

	Prevention & Detection Measures						
	A	B	C	D	E	F	G
Defect# 1	X	X		X			
Defect# 2		X					X
Defect# 3	X						
Defect# 4	X	X			X		
Defect# 5	X					X	
Defect# 6			X				

Significant Overlap
(Note: Four defects are prevented by using prevention measures A & B)

Figure 7. Effectiveness of combining process changes

Future plans for the database tool will allow the user to use estimates of the expected effectiveness of a particular process change on each individual defect and to use information on expected cost to implement the process change in the next release (the present database tool assumes that all process changes are 100% effective and have zero cost). The database tool identifies the combination of process changes that will eliminate the greatest number of defects for a given budget for implementing process changes. Analysis of selected process changes consisting of combinations

of both prevention and detection measures can be performed manually with the tool. Text and graphics generated by the tool can be used in written reports.

The **report template** provides a consistent way of presenting the Defect Analysis results and the benefits of implementing the changes recommended. The report template presents the recommended process changes in a way that is easy to understand and shows the potential quality improvement and the cost savings that could be obtained. The Defect Analysis report has two purposes:

1) to inform management of the recommended process changes and their costs and expected benefits, and
2) to provide enough information for a second party to implement the recommended process changes.

III. Experiences Performing Defect Analysis

A. Benefits From the Analysis

A complete cycle of defect analysis has been conducted internally by Bellcore on seven projects; each project analyzed between 20 to 50 defects. We observed some intrinsic benefits to performing defect analyses. Members of the Bellcore Defect Analysis Teams increased their awareness of the software engineering principles. They also became advocates for process improvements in Bellcore.

In the past, software engineering or quality assurance advocates always preached process improvements without considering what actual errors occurred. The process changes recommended in this manner were not always accepted because of the lack of consideration of what actually caused the faults. Defect analysis methodology identifies the process changes that prevent specific field faults in the product. Therefore,the defect analysis methodology is gaining support from development organizations in Bellcore.

The Bellcore Defect Analysis Team's recommendations are usually well accepted because the Team members are a part of the development team. Their opinions are trusted because they are familiar with the development process and not simply speaking from a set of ideals.

If the recommended process changes were implemented, the projects could have seen a significant reduction of field defects, based on the opinions of the Bellcore Defect Analysis Teams. The number of field defects would have

been 25% to 30% lower if we assume that the process changes were 33% effective[1] in preventing the defects.

There could be significant savings resulting from the defect analysis and implementation of the recommended process changes. Because the effort of analyzing field defects is also significant, there is a desire to analyze the cost/benefit of performing defect analysis. However, no data is available on the cost to implement the process changes and on the savings to the user when fewer software problems are encountered. Using available data on the cost to analyze the defects and the cost to fix field defects, the potential savings from the reduced field defects typically is one to two times the cost of performing the Defect Analysis.

The entire defect analysis process for each new project takes roughly 20 to 30 meetings which range from two to five hours. It is expected that the effort would be reduced if the projects continue with defect analysis on an ongoing basis because the groundwork activities would not require as much effort.

B. Process Similarities

In preparation of this paper, we analyzed the raw Defect Analysis data from the seven Bellcore projects to determine if we can conclude that their defect distributions are similar. Using a simple χ^2 test, we concluded that not all the projects are similar in terms of their fault distribution. With this, we cannot compare our fault distribution to the finding that Ramamoorthy has published (i.e., almost half of all software errors are introduced during the requirements phase of the development life cycle) [10].

Pareto analysis of the process changes does indicate some similarities in the recommendations. Out of a total of more than 90 different prevention measures which can be chosen, there were a handful of process changes that were consistently chosen by the Bellcore Defect Analysis Teams in their recommendations.

Five projects have chosen to recommend improvements to their existing detailed design procedures. Four Bellcore projects chose to recommend improvements to their existing system design guidelines or procedures. All seven Bellcore projects recommended improvements to their design procedures.

---

1 Assuming a 33% level of effectiveness for software process changes is conservative. R. G. Mays [8] of IBM has reported that software development process changes can range from 30 - 70% effective. Data collected and reported by Boehm and Thayer [3] is in general agreement with this.

A checklist that identifies the errors that would most probably occur is another item that is frequently recommended by Bellcore Defect Analysis Teams. Two projects recommended checklists for the system design phase. Four projects recommended checklists for the detailed design phase. And, four projects recommended checklists for the development and coding phase.

The most frequently recommended change in the functional requirement phase is related to communications between developers, user consultants, system engineers, and system testers. Four Bellcore projects recommended obtaining user consultant input, obtaining key person input, or having regular meetings with members of the development team,.

Five Bellcore projects recommended formalizing their unit test procedures to either including a demonstration, test case review, or test results documentation. Five projects recommended improvements to their project code review or inspection procedures.

C. Implementing Process Changes

Most projects do not have problems accepting the recommendations from the Defect Analysis Team. The Defect Analysis Team passes the recommendations to their Bellcore Quality Improvement Team and the Quality Improvement Team takes the recommendations, develops action plans, assigns responsibility, and performs tracking of the action items.

Because most Quality Improvement Teams already had projects they were working on, the recommendations are not getting implemented as soon as the Bellcore Defect Analysis Teams would like to see. Another problem the authors observed is that there is no mechanism to track the effectiveness of the process changes. Ideally, if there is a measurement of errors at the end of each phase, Defect Analysis Team should be able to determine if there is a change in the error counts or fault distribution of a particular project.

D. Performing Defect Analysis on an ongoing basis

The initial effort to perform defect analysis is laborious and tedious. A good way to build a defect database is to analyze small numbers of software problem reports over a long period of time. As the defect data accumulates, the results of the analysis become more meaningful. Tools like the design checklist and the coding checklist can be revised with the ongoing defect analysis of software problem reports. Historical defect data can be used to determine if an organization's preventative measures have been effective. Over a period of time the original class of problems should diminish only to be replaced by a new class of problems. If the same problems continue to

appear, then the organization under review may need to review the preventative measures that were put in place and take corrective actions.

Some Bellcore projects after having completed the first round of analysis, have begun the analysis of defects on an ongoing basis. These projects meet regularly about every two to three weeks to analyze the individual defects. Quarterly reports are made on the progress of the team. When there is enough data, the database is analyzed to determine recommendations for process changes.

IV. Conclusions

The Defect Analysis approach to improving software quality and productivity focuses on the elimination of the most frequently occurring and troublesome software defects in the earliest stages of the software life cycle. Conclusions regarding where maximum benefits can be gained are arrived at objectively and are reproducible from the defect data collected and analyzed.

In general, project teams tend to be transitory in nature. The members of the "1st generation" project team (i.e., software product's emerging state) struggle with the requirement writing, designing, implementing, testing and timely delivery of a software product. The members of the "1st generation" project team accumulate experience during the struggle and then usually move on. Defect Analysis provides a repository of "lessons learned" from the "1st generation" project team. Continued defect analysis benefits subsequent project teams and improves the quality of the software.

V. Acknowledgements

The authors would like to recognize Gardner Patton of Bellcore for his contribution of the Defect Analysis Team concept which is used in Defect Analysis. We would like to thank Robert Hausman, Jen Tang, and Robert Thien of Bellcore for their invaluable assistance and input with the statistical analysis techniques used in Defect Analysis and review of this paper.

VI. References

1. Boehm, Barry W., "Improving Software Productivity", IEEE *Computer*, Vol. 20, no. 9, pp 41-57, September 1987

2. Glass, Robert L., Ronald A. Noiseux, *Software Maintainer's Guidebook*, Prentice Hall, Englewood Cliffs, NJ, 1981

3. Boehm, Barry W., *Software Engineering Economics*, McGraw Hill Book Company, New York, NY, 1981

4. Humphery, W.S., "Characterizing the Software Process: A Maturity Framework", IEEE Software, pp 73-79, March 1988.

5. TR-TSY-000179, *Software Quality Program Generic Requirements*, Issue 1, July,1989

6. Jones, C.L., "A Process-Integrated Approach to Defect Prevention", *IBM Systems Journal*, Vol. 24, no. 2, pp 150-167, 1985

7. Ryan, J., "This Company Hates Surprises", ASQC *Quality Progress*, Sept 1987

8. Mays, R. G., C. L. Jones, G. J. Holloway, and D. P. Studinsky, Experiences With Defect Prevention", *IBM Systems Journal*, Vol 29, no. 1, pp 4-32, 1990

9. Fagan, M.E., Design and Code Inspections to Reduce Errors in Program Development, *IBM Systems Journal*, Vol. 15 no. 3, pp 182 - 211, 1976

10. Ramamoorthy, et. al. "Software Engineering." IEEE *Computer*, October 1984.

Methods and Mechanics of Creating Verifiable User Documentation

Andrew Oram

August 2, 1991

Hitachi Computer Products (America), Inc.
Open Systems Software Development
Reservoir Place
1601 Trapelo Road
Waltham, Mass. 02154

E-mail: uunet!hicomb!oram, hicomb!oram@uunet.uu.net

Abstract

User documentation, long considered an unwelcome responsibility for software project teams, can actually be produced with the same processes of specification, verification, and measurement as the other deliverables in a computer system. This paper describes a practical, inexpensive method that some commercial computer vendors are using to create and review their manuals. It uses a simple form of constructive specification to determine the valid operations that users can perform. The method leads to a set of usage models and a series of examples that can be integrated into automatic regression tests. Benefits include better documentation of environmental needs such as prerequisites and restrictions, clear links between user tasks and product features, and regular automatic checks on the document's accuracy.

Biography

Andrew Oram has written manuals about network administration and programming, array processors, compilers and debuggers, and real-time computer systems, and has developed regression tests and standards covering documentation for Quality Assurance teams. Recent projects include programmer's documentation for the Hitachi port of the OSF/1 operating system, and a revision of *Managing Projects with make* for O'Reilly & Associates.

The software field has long held an ambivalent attitude toward user documentation. Programmers and Quality Assurance staff definitely appreciate when a good manual helps them learn about their own projects. And in the software engineering literature, one would have difficulty finding a text that fails to list user documentation as a deliverable. But on the other hand, engineers do not feel comfortable with specifications and evaluation in the area of user documentation. Thus, many relegate it to the fringes of their projects, sometimes tacking it on at the last minute. Ironically, the trend in software engineering literature [such as Boehm, 1981; ANSI/IEEE, 1986] is toward the other extreme — to treat user documentation as part of software requirements, and thus to insist unrealistically that it be largely finished before software design even begins.

This paper tries to bring the little-researched area of user documentation within the software engineering fold. I will describe a practical, inexpensive method that some commercial computer vendors are using to verify and monitor their manuals. Software project managers, designers, and Quality Assurance staff can use the method to extract the verifiable elements of documentation and work them into specifications, test plans, and schedules.

The stage in this method resemble the informal techniques that many people use when they have to document software — roughly:

1. Decide what features to discuss.
2. Play with sample applications in order to learn how the software works.
3. Organize the models, procedures, warnings, and other insights into a reasonable sequence.

The contribution of this paper is to give these techniques a firm grounding in software engineering. This makes the difference between an unstructured play activity and a discipline that supports goal setting and resource allocation (without losing any of the fun).

In pursuit of verifiability, this paper defines exactly what a "feature" is, and offers a complete list of questions that have to be answered in order to document each feature. I also show how to determine that the applications discussed are truly of value, and how to associate the models offered to readers with the actual steps they must follow to use the software. Every stage of the method includes rules for bounding the activity, recording progress, and reviewing results.

Before I launch into the theoretical underpinnings, let me describe some incidents that give the flavor of what it is like to work with this method in a commercial environment.

- During the development of a real-time operating system, the examples from one manual were tested on systems with up to four processors and judged correct. However, during the next release of the product, the test team obtained an eight-processor system, reran the regression suites, and found a subtle omission in one of the examples. It took less than an hour to correct both the example and the explanatory text in the manual.
- In the documentation effort for parallel-processing tools, a relatively trivial example turned up an internal compiler error in the assignment of classes to variables. The error had never been found by regular QA tests because one class was rarely used, and because the tester focused on stress testing with unrealistic programming constructs. The documentation example performed a more realistic operation mixing data of different classes, and thus triggered the error.
- On a project in data base administration, the writer analyzed the material to find the first task an administrator would be likely to perform. She discovered that the required knowledge for this task (initializing the user security information) was currently divided among three separate manuals. In a few months, she produced a tutorial that covered this basic task and several others in simple, procedural fashion.

User documentation is the culmination of a long process of discussion and experimentation throughout a software project. Therefore, while this paper's main impetus is to foster better publications and on-line documentation for users, some of its recommendations will affect a project's internal documentation and staff training. Thus, the paper should interest people concerned with improving education and communication among their programming staff, and particularly with ways to disseminate the insights of project designers and senior members to other people on the team.

The next three sections — **GOALS, THEORY,** and **ROLES** — show the method's general fitness for software documentation. The bulk of the paper is devoted to a history of practical applications: a stage-by-stage description in **METHOD,** and a discussion of implementation details in **MECHANICS.** I end with a summary of the method's current status in **BENEFITS.**

GOALS

The traits of good documentation that can be developed through a rigorous method are:

- To list the prerequisites for each feature.
- To illustrate each feature of the product in a context recognizable to readers.
- To lead the reader in small steps from simple uses to complex uses.
- To cross-reference between high-level concepts (like user tasks and programming models) and lower-level concepts (like system prerequisites and product features).
- To reflect changes in the product through its lifetime.

By way of contrast, here are some traits that cannot be verified formally, but depend on the individual skill and subjective judgement of the document's producers — and therefore, lie outside our discussion.

- To use terms appropriate for its readers.
- To offer the right amount of general background information.
- To use a format and a layout on the page or screen that it is easy to follow.
- To remain free of typographical errors and other problems that lie in the gap between the material that is verifiable and the actual printed material.

The verifiable traits of documentation are essentially linked to features of the product and its use, while the non-verifiable traits cover the psychological aspects of the document and its translation into a medium of distribution.

Desirable results are not enough to define a useful working method. The implementation must also be feasible in a commercial environment. Thus, a method to produce verifiable documentation should meet the following procedural requirements.

- Repeatable use.

  The verifiable traits of the documentation can be checked through regression tests at regular points in the software's development cycle.

- Low cost.

  The method adds a relatively small burden to the existing responsibilities, schedule, and computer resources of a commercial project team.

- Scalability.

  The same essential techniques can benefit small projects (such as one-person MIS projects directed toward a few in-house users) as well as large ones (commercial software for end-users, where the documentation is the critical entry point to the product).

- Ease of integration and knowledge transfer.

  The techniques ring familiar to well-trained engineers and Quality Assurance staff, and can be adapted to whatever standards they are using for other software maintenance efforts.

- Value for retroactive use.

  The method can be used to produce documentation for software that has already been released, and even software whose original designer has left or whose project team has disbanded.

THEORY

Other articles [Oram, 1989; Oram, 1991] have laid out and justified the underlying thesis for verifiable documentation:

The critical issues determining the quality of software documentation lie in the structure of the software itself, not in stylistic choices made by the writer.

This paper will show that one can produce a complete description of a system's use by tracing data transformations from one function to the next. The supporting theory for our endeavor is constructive specification. It may seem a surprising choice, since the theory is best known as a somewhat academic, labor-intensive method for constructing formal proofs [Jones, 1980] and as a way of deriving classes in object-oriented programming [Stoy, 1982]. But in this paper, constructive specification proves to be a simple and powerful way to link software's use with its logical structure.

The basic idea behind constructive specification is to describe every data object in terms of the operations that the program will allow. For instance, you can write a specification for a stack by describing three operations: initializing, pushing, and popping. For the purposes of documentation, we can set both a direction and a boundary to our efforts through the following rule:

The specification of a user document is complete when it includes every operation that is valid on every data object that affects system state, within a sample application that causes a change from one user-recognizable system state to another.

Let us now decipher the key phrases "every data object that affects system state" and "user-recognizable system state." We can then join the theory to a broader view of mental models and links.

Data Affecting Product Use

Data that pertains to user documentation falls into two categories: function arguments and static data (which includes data stored in external media). One benefit of developing sample applications is that they help to reveal the user's dependence on state information. A document based on examples is unlikely to leave out critical prerequisites or environmental needs, such as to mount a disk, to reserve unusually large buffers, or to log in with special privileges. All these environmental needs (loosely related to the concept of "non-functional requirements" in [Roman, 1985]) are stored as the internal static data mentioned earlier.

Sample Applications

Applications, according to the rule stated early, should causes changes between "user-recognizable" states. This stipulation is meant to rule out dummy examples, like one that simply converts an integer to a character string. By contrast, a simple example of calculation and report generation that include conversion between an integer and a character string can be a valuable teaching tool, because the conversion now becomes part of a larger task and is justified by the requirements of that task. Because such an example is anchored by useful, recognizable states at both the beginning and the end, I have coined the term *portal-to-portal verification* to describe it.

Sample applications can emerge through both top-down and bottom-up approaches. The top-down approach, which is the more familiar one, consists of collecting benchmarks and customer applications that the product is meant to support, and breaking them down into small pieces that are independently verifiable. But this cannot ensure full coverage of all operations on all data items. Thus, it must be accompanied by the bottom-up approach, which is to trace data transformations using the method in this paper.

Here is a simple example of bottom-up design. The basic operations on a file identifier include assignment (through an **open** statement), reference (through **read**, **write**, and **close** statements), and ancillary operations (like FORTRAN's INQUIRE or the C language's **stat**). Thus, one can create a simple portal-to-portal example by opening, writing, and closing a file. Verification could consist of comparing the resulting file to a canned version, or of reading the data back into the program and checking it for consistency.

Simple as such an example is, the lessons it embodies are by no means trivial. It can be the template for sophisticated applications like imposing structures on raw binary data, and opening a pipe with non-blocking (asynchronous) access. Using the method in this paper, one can build a complete description of file handling through a series of progressively more complex examples.

This paper is the first, to my knowledge, to suggest a disciplined method using examples to assure full product coverage. I have found only one other discussion of user examples in the software engineering literature [Probert, 1984] but it considers them a source for tests rather than a training tool.

Software Structure and Models for Use

One goal of computer documentation is to identify user tasks. But the defining characteristic of a "task," in the area of computer software, is that it really consists of many tasks on different conceptual levels.

For instance, in a relational database, users might define their task initially as retrieving the entries that match certain criteria. But to begin using a typical query system, they have to redefine this task as "building a view." This task in turn depends on a lower layer of tasks like choosing the keys to search for, creating Boolean search expressions, and sorting the entries. The documentation discussed in this paper helps users develop the necessary thought processes for figuring out how to use the software — that is, for decomposing their tasks until they reach the atoms represented by the product's features.

Cognitive scientists and educators have focused on the concept of mental models to explain how people assimilate information and apply it in new situations. The more sophisticated research [for instance, Brown, 1986; Norman, 1986; Frese, 1988] bolsters the strategy used in this paper: that of matching the models of product use to the logical structure of the software.

Verifiable documentation builds models from the structure of product itself, which offers both richness and accuracy. The models are simply the uppermost layer of user tasks, such as "searching" in the example of a data base. The user who consults the documentation in order to perform a search finds a progressive break-down into lower levels of tasks, ending perhaps in the arguments of a WHERE clause in SQL.

In this paper's method, models map directly onto the designer's construction of the software. For example, a real-time programming manual could divide applications into *cyclic* and *interrupt-driven.* Cyclic applications could then be broken down further into those running several independent threads, and those running several functions repeatedly in one thread. The manual can then describe the environments in which each model would be most advantageous, and implement each model through procedures and examples.

Links and Document Structure

For initial learning purposes, users tend to read in a linear manner (even though they also tend to start in the middle and skip around a lot). For reference and trouble-shooting purposes, they prefer to search a hierarchical structure, such as an index or an on-line set of hypertext links.

No one uses every feature in a product. But one can be certain that certain sets of users need particular combinations of features. Thus, I use the metaphor of *terraces* to describe the structure of a computer document. Each terrace consists of an example with its accompanying explanation. A document can have many "hills," each consisting of a set of terraces that increases gradually in complexity.

Thus, in a database product, one hill could offer more and more complicated examples of retrieving keys, thus showing the reader various ways to build a view. Another hill could solve the problems of physically storing large databases. Users can climb the hills that they need for their particular applications, and ignore other hills entirely. If new features or new applications are added during product development, the writer can find places for them near the top of the terrace hierarchy. But the disciplined creation of sample applications ensures that users can associate tasks with product features.

ROLES

The main goal of this paper is a good document to be delivered to the users. But its philosophy applies to internal project documentation too. Broadly speaking, this paper asks:

How do software designers convey their insights to the less experienced team members during product implementation, and ultimately, in manuals and training courses, to the end-users of the product?

Like any planning strategy, the method in this paper is least expensive and most beneficial when it is employed from the earliest phases of a project. The managers who initiate the project can, with fairly little effort, preserve some of the user applications driving the project in the concepts and requirements documentation.

Software designers definitely have usage models in mind as they find common sub-tasks, create modules, and define system-wide data structures. The models should be explicitly documented in the software design descriptions. If the designers do not have time to create full examples, they can delegate the work to other team members — in either case, the intellectual process of creating examples helps to define the product and describe it to the team.

The method in this paper is equally valuable in the unhappy — but all too common — situation where a product has been in the field for a long time without adequate documentation, and the project team hires a writer to redress the situation. Now the method provides guidance for reconstructing the lost information on use. Categorizing and tracing the data helps to establish essential information, like what each command option is for, and what distinguishes similar commands. Where features cannot be understood, and further research on user applications is needed, the method helps the writer identify missing information and pose the right questions.

METHOD

In a commercial environment, the production of verifiable documentation falls into three distinct stages. While this paper discusses them sequentially for the sake of simplicity, the pressures of real-life project development often force variations.

For instance, on a project with tight deadlines, some stages overlap in a pipeline. A partially-completed data analysis can be used to start developing examples, and early sets of examples can be placed in a tentative order so that the writer can start creating the text.

Changes in design or marketing strategy also complicate the method by requiring the team to reiterate completed stages. If a new feature is added, each of the documents produced in each stage must be adjusted to include the feature. One of the method's strengths is that writers can quickly determine the ripple effect of any change on the entire user viewpoint, and make incremental changes to documentation where necessary.

Stage 1: List and Categorize all Data Items Affecting the User

In the section on **THEORY**, I described the input and system state data to which product designers should be alert, and showed how the use of the product is fully determined by this data. The first stage of this paper's method categorizes data items, determining the role that each plays in the software system.

Activities: For the purposes of user documentation, a few simple categories suffice to accommodate all data items. These appear in Figure 1, along with rules for determining the proper category for each item. Categorizations might not be intuitively obvious when the user operates at a level far removed from the engineers (as in the case of interactive graphics products) but even so, the same categories always apply. One must get used to looking at the object's role in the state of the system, rather than looking at the superficial mechanisms for selecting or modifying an object.

Category	Programming level	Command level	Menu level
Flag	A Boolean variable, one that is permanently restricted to having two possible settings.	An option that is either present or absent, with no accompanying value.	An option that the user chooses without entering any value, choosing from a list, etc.
Counter	An integer (generally unsigned) that is incremented or decremented during the course of the application, and is checked for reaching a threshold (often zero).		
Identifier	A value (usually integer or string) that is assigned at most once, is never changed thereafter, and is referred to in order to locate the object. This category of data covers file descriptors, channel identifiers, and other objects offered by the operating system.		
Table	An integer used as an offset or array subscript (assuming that the array is a set of mappings or pointers, rather than a vector with arbitrary contents).	An option accepting a fixed set of values, usually represented as character strings.	A menu of options, from which the user chooses exactly one.
Application data	Any item other than a counter that can take a range of values, with no fixed, predefined set of possible values. Examples include file names and the data entered into a spreadsheet.		

Figure 1. How to Categorize Data Items.

Some people might want to use these categorizations as primitives from which to derive more specific categories. For instance, a file identifier, a process identifier, and a channel identifier all support different data transformations. Thus, if your product has many data items that fall such sub-categories, you might find it efficient to create separate lists of questions for each sub-category.

Similarly, some data items cover more than one category. For instance, a communications protocol might define several possible encodings for a single data item, where the settings of certain bits determine which encoding applies. Many UNIX and X Window System applications use C-language unions for similar purposes. Such complexities merely mean that you have to ask the appropriate questions for each possible use of the data item.

Since the focus here is on the data's purpose, we do not need to be concerned with its type, scope, or range. (These considerations do of course appear eventually in the documentation, to describe restrictions and error conditions.) Nor are derived data types or levels of indirection important; we are concerned only with the kinds of transformations allowed.

Once a category is found for each data item, we know the information that the documentation must provide for that item. The list of questions for each category appears in Figure 2.

Flag	a. Default setting. b. Who sets it, why, and how. c. Who resets (clears) it, why, and how. d. Who reads it, why, and how.
Counter	a. Who initializes it, why, and how. b. Who increments or decrements it, why, and how. c. Who reads it, why, and how.
Identifier	a. What object it refers to. b. Who initializes it, and how. c. Who refers to it, and what operations are generally performed on the object. d. Whether it is ever deleted, and how.
Table	What each entry in the table is for, and how to select it (at least one example per mapping within the table).
Application data	a. What the default is, and why use it. b. How a value is used. c. Special considerations. These occur most often at the edges of the range. For instance, zero or null often has a special meaning that cannot be intuitively extrapolated from the normal range of values.

Figure 2. What Must be Documented for Each Data Item.

Result: Stage 1 achieves two major steps that are required to manage any activity. First, by breaking down a documentation effort that up to now has been undifferentiated, this stage allows the team to prioritize sub-tasks and assign them to different team members. Second, by providing a complete list of sub-tasks, it provides an admittedly crude but still useful guide toward making documentation's progress measurable.

The concrete result of Stage 1 is a list like Figure 3. This figure is excerpted from an actual internal document developed during the design of a programming product for window graphics. The left column is simply a list of functions, while the next column shows each function's argument list. If the product included state data, these could be listed in the left column as well.

Call	Argument/ Data item	Categorization	Expected range	Examples
Init( )	return value	table	True (success) False (failure), BadAccess	
	display	identifier	retrieved from server	
ChangeScheduler( )	display	identifier	retrieved from server	
	client	identifier	from ThisClient( )	
	when	table	Immediate Sequential	
	params:			
	type	table	RoundRobin PriorityBased	
	slicevalue	counter	> 0	
	slicetype	table	TimeBased RequestBased	
	decaylevel	counter	MinPriority to MaxPriority	
	decayfreq	counter	> 0	
	decayunits	table	TimeBased RequestBased	
	priority	counter	MinPriority to MaxPriority	
	priomode	table	Absolute Relative	
		.		
		.		
		.		

Figure 3. Sample Internal Document Generated by Stage 1 — Window Product.

The argument to one call is a complicated structure containing several distinct data items. Figure 3 reflects this by indenting the data items under the name of the structure, **params**.

Some data items now require more than one row, because multiple settings must be documented. In particular:

- Flags require two rows.
- Counters and application data sometimes require extra rows for values with special meanings (such as negative values or zero).
- Tables require one row for each legal value.

The **Categorization** column reflects the criteria from Figure 1. The **Expected range** column resembles the domain and range information collected in standard Quality Assurance practices. In general, the third and fourth columns embody the strategy for exploring the product and answering the questions in Figure 2.

The rightmost column is currently empty, but will be filled in during Stage 2 with a list of examples that illustrate each particular data item at each specified value.

As another example of Stage 1 output for more familiar software, Figure 4 shows the data categorization for the *signal* call on UNIX systems (as standardized in ANSI C).

Call	Argument/ Data item	Categorization	Expected range	Examples
signal	signo	identifier	mnemonic for signal	—
	sa_handler	table entry	SIG_DFL	—
			SIG_IGN	—
			function	—
	return value	table entry	SIG_DFL	—
			SIG_IGN	—
			function	—
			SIG_ERR	—

Figure 4. Sample Internal Document Generated by Stage 1 — Signals.

Review: This stage produces a small amount of output, tied closely to the software design. Reviewers should examine the output to ensure that:

1. All function arguments and static data are listed, so long as these could have an effect on product use.
2. Each data item is correctly categorized.

Analogies within software engineering: In software engineering terms, the goal of this stage can be compared to *acceptance criteria.*

Analogies in technical writing: In terms used by the technical writing literature, this stage meets the goal of *comprehensiveness.*

Stage 2: Develop Examples

In Stage 1, the method delved deeply into the internal logic of the product. The result was a history of changes to each data item. It is time now to return upward to the user's point of view.

Stage 2 builds the changes for each data item into small but realistic applications. This stage contains the alchemy that transforms system states into user tasks and programming models.

Activities: To some extent, examples emerge naturally from the answers to the questions in Figure 2. For instance, in asking "What function assigns the identifier that appears as this argument?" one discovers the kinds of initialization required in a sample application. While following the lines of data transformation, one discovers functions that go together naturally. Eventually, one or two rock-bottom examples emerge as the simplest possible applications that use the product to do something constructive.

The achievement of Stage 2 is to link product features to progressively higher layers of tasks. The links collectively form a cross-reference system that the writer can turn into an index for a manual, or a set of links in on-line documentation.

Example-building is a bounded activity, because the previous stages have already defined what data items must be documented, and what questions the documentation must answer for each one. The success of the documentation effort is now quantifiable. If time does not permit the full exploration of every data item, the engineering team can choose to focus on critical items and ignore obscure ones.

Result: The language or medium for examples depends on the type of product being documented. Functions and programming languages are illustrated by examples of code. Operating systems and interactive utilities can be illustrated by series of prompts and commands. Point-and-click applications call for pictures or descriptions of the mouse and keyboard movements.

In the window project discussed earlier, the search for examples radically altered the document's focus. No meaningful application could be developed that stayed within the scope of the product. Instead, the team agreed to pull in numerous tasks that lay outside the software they were building, but which were an inseparable part of the application base for the software:

- Control over the total windowing environment, which could contain any number of unrelated applications.
- Communication and synchronization using the channels provided by the operating system.
- Responses to real-time input, of both periodic and urgent forms.

The initial document created for review during Stage 2 looked like the pseudo-code in Figure 5. The final document was a set of actual programs. A subset of Figure 5 is represented by the C code in Figure 6, which is an excerpt from an actual program testing features from the product. As with software testing, a complete set of user examples should include some that are supposed to fail, by deliberately causing errors or breaking the documented rules.

Basic example of changing the scheduling policy, when user presses button

External event	Program action
	Init(display) . . . client_id[0] = ThisClient(display) params.type = PriorityBased params_raise.type = PriorityBased params.u.p.slicevalue = long_draw + 10 params_raise.u.p.slicevalue = long_draw + 10 . . . ChangeScheduler(display, client_id[0], &params, Sequential)
Button press	for (i=0; i<num_clients; i++) ChangeScheduler(display, client_id[i], &params_raise, Immediate) . . .

Figure 5. Sample Internal Document Generated by Stage 2.

```
#include "plot_xlib.h"
#include "root_defs.h"

void Xserver_priority_initialize(top)
   DISPLAY_INFO *top;
{
 rtxParms set_rtxp;

 /* initialize with privileges to change global parameters */
 (void) PrivilegeInit(top->display);

 /* client-specific parameters will affect just this client --
   actually, this fragment affects only global parameters */
 top->client = ThisClient(top->display);

 /* this will tell library to check the set_rtxp.u.p parameters */
 set_rtxp.type = PriorityBased;

 /* mask makes scheduler change slice and decay, but leave priority alone */
 set_rtxp.mask = Slice | DecayLevel | DecayAmount | DecayFrequency;

 /* set the parameters of the rtxParms structure */
 set_rtxp.u.p.slicevalue = NEW_SLICE;
 set_rtxp.u.p.dlevel = CEILING_FOR_DECAY;
 set_rtxp.u.p.damount = 1;
 set_rtxp.u.p.decayfreq = DECAY_TIME;
 set_rtxp.u.p.slicetype = TimeBased;
 set_rtxp.u.p.decayunits = TimeBased;

 /* Everything before was prepartion -- this call makes the change */
 ChangeScheduler(top->display, top->client, &set_rtxp, Sequential);
}
```

Figure 6. Function Excerpted from Sample Program Generated by Stage 2.

Review: Since this stage produces a great deal of output, it is often carried out incrementally. The primary criteria are:

- Correct use of each function or command.
- Adherence to pre-requisites, with correct set-up and clean-up activities.
- Complete coverage of the usage models defined by senior project members and project managers.
- Correct delineation of tasks under each usage model.

Some additional optional criteria can help to improve the quality of the final documentation or the maintenance effort for examples. These criteria require intuitive judgement and a sense of the user environment.

- Adherence to standard, recommended practices in areas outside the features of the product.
- Optimality. Each example should be smallest and simplest one that could illustrate the use of the data, while still maintaining some naturalism.
- Usefulness. All other things being equal, it would be valuable to build sets of examples that approach realistic applications. However, users and applications evolve unpredictably, so it is much more important for examples to be simple and convey the basic use of the product.
- Machine independence. Regression testing is compromised if the example implicitly assumes a certain underlying architecture, directory structure, or other external elements that are known to change over time.
- Ease of testing. For instance, an example that generates data internally or uses pre-defined input requires a lot less work during regression testing than one that interacts with a user for its input.

Analogies within software engineering: The goal of this stage can be compared to developing *test suites.* In fact, the examples should be integrated into regression tests, as discussed in the **MECHANICS** section of this paper. During product design, examples validate the documentation effort by ensuring that it supports the necessary user applications and styles. The regression tests, in turn, verify that the examples reflect product operation over the entire life of the product.

Analogies in technical writing: In technical writing terms, the decomposition of user tasks fulfills the goals of *usability* and *task orientation.*

Stage 3: Order the Examples

This stage organizes the examples of Stage 2 into a structure that determines the order of presentation in the final document. The structure has elements of both a simple linear ordering and a tree hierarchy. The linear elements will be more evident in a printed manual, and the hierarchical elements in on-line documentation.

Activities: Stage 3 continues the movement started in Stage 2, away from the structure of the software and more toward the human user. The goals are to make it as easy as possible to get started with a product, and then to identify and incorporate useful enhancements. Examples provide natural criteria for ordering information in a linear fashion:

- Simple applications before complex ones. Start with the simplest possible example of product use, and try to introduce only one or two new features in each successive example.
- Common features before obscure ones. Special options that fulfill rare needs can be isolated near the back of the document.

The hierarchical aspects of organization come from the models and tasks discussed under Stage 2. These help to group together the examples that users need at a particular time.

Result: The result of Stage 3 is a re-ordered set of examples organized under a hierarchy of tasks. Since it represents the final structure of the manual, reviewers can examine the hierarchy to decide whether features are presented in an appropriate order and given the right amount of attention.

At the end of this stage, the engineering team has formally defined both the topics and the structure of the product's documentation. The richness and authority of the information that this resource offers to technical writers cannot be matched by any other method. Writers can now prepare background information and narrative text that explains the models, tasks, and techniques. The cross-referencing system can be used to build the index.

As a brief example of a document structure designed through data analysis, here is the outline for an on-line, fully task-oriented description of ANSI C and POSIX signals [ANSI, 1989; IEEE, 1988] .

- Trapping
  - Basic *signal* call
  - POSIX signals (*sigaction*)
    - Definition
    - Flags
    - Errors
  - Forks (inheriting signal actions)
- Handlers
- Sending
  - *kill* call
    - Basic example
    - Processes/groups
    - Privilege
    - Errors
  - Interprocess communication
  - *raise* call
  - *alarm* call
- Blocking
  - Signal set (manipulating the *sigset_t* data type)
  - Data protection (*sigprocmask* call)
  - Handler protection (in *sigaction* call)
  - Pending signals (*sigpending* call, *sigismember* call)
- Waiting for signal (*sigsuspend* call)

The document moves from simple issues to more complex ones, freely breaking up the discussion of a single call where task-orientation calls for it. For instance, the section on "Sending" focuses on communication with a single process (the most common case), but also offers a brief discussion of the more complicated issue of process groups. Although blocking is a critical issue, it comes late in the document because it requires a good understanding of the earlier issues. Many issues not directly related to the calls also appear in the document, such as the need to pass information between the handler and the main program using volatile, atomic data.

Review: The review of Stage 2, if carried out thoroughly, has resolved the most important documentation questions. Reviewers should be able to accept at Stage 3 that the models, tasks, and examples are the best available. Thus, the review at Stage 3 ascertains whether:

- The linear organization is best available, in terms of putting the simpler and more common applications first.
- The hierarchical organization links all the layers of models, tasks, and features established in Stage 2.

After this stage — when the materials are in the hands of the writers — the focus moves to reviewing the text in relation to the examples. Document review becomes much easier and more rewarding, because it can focus on small areas of the document and ask questions whose answers are fairly easy to determine: for instance, whether the written procedures accurately summarize the examples, and whether the text warns users about potential sources of error.

Analogies within software engineering: This stage does not have a real counterpart in software engineering, because programs are generally not linear texts.

Analogies in technical writing: In technical writing terms, this stage meets the design goals of *structured documentation*, by filtering and pacing information for easy learning and retrieval.

MECHANICS

This section shows some the tools and organizational structures that my colleagues and I have used to integrate examples into regression tests. This part of the verification effort has offered the most rewards in relation to invested time and resources.

A simple example is furnished by a book on the UNIX system's *make* utility. The data analysis included all command options, in particular an **-n** option that causes the utility to print a series of commands.

An interesting test for the **-n** option is a set of nested or recursive *make* commands. First, create a file named *makefile* with specifications for *make*. The following is simplified but still realistic example.

```
all :
    ${MAKE} enter testex

enter : parse.o find_token.o global.o
    ${CC} -o $@ parse.o find_token.o global.o

testex : parse.o find_token.o global.o interact.o
    ${CC} -o $@ parse.o find_token.o global.o interact.o
```

Interactively, one can test the **-n** option by entering the command:

```
make -n all
```

which should produce output like:

```
make enter testex
cc -O -c parse.c
cc -O -c find_token.c
cc -O -c global.c
cc -o enter parse.o find_token.o global.o
cc -O -c interact.c
cc -o testex parse.o find_token.o global.o interact.o
```

While some output lines vary from system to system, others are reasonably predictable. Thus, one could begin automating the test by redirecting the output to a file, and then checking to see whether one of the lines is correct:

```
make -n all > /tmp/make_n_option$$
grep 'cc -o enter parse.o find_token.o global.o' /tmp/make_n_option$$
```

Finally, we can put the whole sequence into a regression test by running it as a shell script. Figure 7 shows the final result. For readers who are unfamiliar with shell scripts, I will simply say that the following one is driven by an invisible exit status returned by each command.

```
rm -f *.o enter testex
if
  make -n all > /tmp/make_n_option$$
then
  if
    grep 'cc -o enter parse.o find_token.o global.o' /tmp/make_n_option$$
  then
    exitstat=0
  else
    echo 'make -n did not correctly echo commands from recursive make'
    exitstat=1
  fi
else
  echo 'make -n exited with error: check accuracy of this test'
  exitstat=2
fi
rm -f /tmp/make_n_option$$
exit $exitstat
```

Figure 7. Automated Test.

An interesting sidelight from this example is that it reveals incompatibilities among UNIX systems. While the test uses entirely standard, documented features, some variants of *make* have not implemented them.

Figure 8 shows another style of test automation, through a short example from a section of a FORTRAN manual on parallel processing. The manual includes marginal comments explaining the procedure, which to fill an array in parallel through a loop.

```
      REAL A, B, ARRAY(100), TMPA, TMPB
C
      PRINT*, 'Input two reals: '
      READ (*,*) A, B

CPAR$ PARALLEL PDO     NEW(TMPA, TMPB)
CPAR$ INITIAL SECTION
      TMPA = A
      TMPB = B
CPAR$ END INITIAL SECTION
      DO 20 I = 1, 100
        ARRAY(I) = ARRAY(I)/(SIN(TMPA)*TMPB + COS(TMPB)*TMPA)
 20   CONTINUE
```

Figure 8. Programming Example as it Appears in the Manual.

Figure 9 shows the example augmented by Quality Assurance staff to be self-testing. The figure does not include the long header comments contained in the actual test, to describe the purpose of the example and include a simple shell script for running and verifying it. The ARRAYVFY array has been added to store comparison data, and the EXITSTAT variable to indicate whether errors have been found. A verification section at the end of the program simply performs the same operation sequentially that the example performed in parallel, and checks the results. Thus, this programming example is completely self-contained. However, the more familiar technique of comparing output against a pre-existing file of correct answers is equally good, and was used by Quality Assurance for some other examples in the same test suite.

```
      REAL A, B, ARRAY(100), TMPA, TMPB
      REAL ARRAYVFY(100)
      INTEGER EXITSTAT /0/

      DO 10 I = 1, 100
      ARRAY(I) = I
      ARRAYVFY(I) = I
 10   CONTINUE

      PRINT*, 'Input two reals:'
      READ (*,*) A, B

CPAR$ PARALLEL PDO     NEW(TMPA, TMPB)
CPAR$ INITIAL SECTION
      TMPA = A
      TMPB = B
CPAR$ END INITIAL SECTION
      DO 20 I = 1, 100
        ARRAY(I) = ARRAY(I)/(SIN(TMPA)*TMPB + COS(TMPB)*TMPA)
 20   CONTINUE
C
C ------ VERIFY ------------------------------------------------
C
      DO 100 I = 1, 100
        ARRAYVFY(I) = ARRAYVFY(I)/(SIN(A)*B + COS(B)*A)
 100  CONTINUE
      DO 200 I = 1, 100
        IF (ARRAY(I) .NE. ARRAYVFY(I)) THEN
           PRINT *, 'Error in array on element I', I,
     &          ARRAY(I), ' <> ', ARRAYVFY(I)
          EXITSTAT = 1
        ENDIF
 200  CONTINUE
      CALL EXIT (1)
      END
```

Figure 9. Programming Example as it Appears in the Regression Test.

To integrate the test into our regression suites, a staff member simply added the source code, and used existing test procedures to compile it, run the program, and check the exit status. I have deliberately shown the primitiveness of our procedures — relying simply on shell scripts and other standard UNIX system tools — to show how low the overhead of test development can be. While the first few tests for each project took a while to create (about one person-hour per documentation example) we soon become familiar with the procedure, and got to the point where we could turn an example into a self-verifying test in about 10 minutes.

Naturally, test development would be easy with more advanced tools. Some of the areas for further research include:

- Folding the documentation analysis in with the creation of the functional specification and the test plan, in order to eliminate duplication of effort. Currently, the method described in this paper is carries on completely separate from the other efforts, and any programs created by those efforts require a great deal of adaptation before being suitable for documentation examples.
- Maintaining a single source of each example for both the regression test and the document. This would require a tool that extracts and formats the portions of the example actually needed in the document. The process is rife with difficulties, and might not be worth the effort. Almost any change to an example requires a corresponding change to the narrative in the document — that is the whole reason for the method in this paper. Therefore, it might be best to keep writers involved and force changes to be transferred to the document manually.
- Automating the transformation from user example (such as Figure 8) to full regression test (Figure 9). Like most automation of software engineering tasks, this is a tricky area.
- Developing hooks in the systems being tested to permit further automation. In real-time programming, for instance, it is very hard to determine whether raising one's priority really results in getting more CPU time. Similarly, it is hard to test a graphics product without manually using the mouse and personally observing the output. These are well-known problems in the computer industry, and extend far beyond the area of user documentation.
- Developing rules that help the team predict areas of failure. This is another classic software engineering dilemma. For instance, one cannot tell whether an example resulted in a corrupted file unless the regression test checks that file.
- Formalizing the assignment of responsibilities. How much example development should be done by software designers, by programmers, and by writers? At what point can these people turn a crude example over to Quality Assurance and say "Now automate it"?

BENEFITS

The method presented in this paper has evolved through numerous projects in which I and my colleagues applied software engineering techniques to user documentation:

- Functions and techniques for controlling window graphics (excerpts of which were used in the **METHOD** section).
- Programming techniques with signals.
- Configuration and testing of OSI and local-area networks.
- Configuration, access control, and query techniques for databases.
- C libraries and FORTRAN language statements that activate parallel processing.
- Real-time programming control over timing, scheduling, processor allocation, and file handling.
- Language debuggers and program-building utilities.

The project on signals was an on-line document, while the rest were hard-copy manuals. Most of the projects involved complex programming tools, which might skew the method. But small experiments producing end-user documentation, as well as the considerations discussed in the **THEORY** section of this paper, suggest that the method can be successful for any audience and any computer product.

Where verifiable documentation has replaced an earlier manual for the same product, comparisons are revealing. The new documents have been generally agreed to display the following benefits:

- They have far more information, while being shorter than their predecessors.
- They expend a far greater amount of space on examples (often 50%), but the sparse narrative information comes out more understandable and relevant.
- They find natural settings and useful applications for complicated features, which earlier documents described in such a confusing and difficult manner that many readers could not make sense of them at all.

The general method for producing verifiable documentation has now reached a fairly stable state, and is well-enough defined to be transferable. As use of the method spreads, I hope to create a community that can develop increasingly sophisticated tools to implement the stages of development, and more research data by which the method can be evaluated. Meanwhile, our practical successes to date, as well as the clear theoretical advance that this method represents over other documentation methods, should make it attractive to software development teams.

References

ANSI, 1989. ANSI, *American National Standard for Information Systems — Programming Language — C*, X3.159-1989, ANSI, New York, NY, 1989.

ANSI/IEEE, 1986. ANSI/IEEE, *IEEE Standard for Software Verification and Validation Plans*, IEEE Std. 1012-1986, IEEE, New York, NY, 1986.

Boehm, 1981. Boehm, Barry W., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

Brown, 1986. Brown, John Seely, "From Cognitive to Social Ergonomics and Beyond," in *User Centered System Design: New Perspectives on Human-Computer Interaction*, ed. Donald A. Norman and Stephen W. Draper, Lawrence Erlbaum Associates, Hillsdale, N.J., 1986, pp. 453-486.

Frese, 1988. Frese, Michael, et al., "The effects of an active development of the mental model in the training process: experimental results in a word processing system," *Behavior and Information Technology*, vol. 7, no. 3, 1988, pp. 295-304.

IEEE, 1988. IEEE, *Portable Operating System Interface for Computer Environments*, IEEE Std. 1003.1-1988, IEEE, New York, NY, 1988.

Jones, 1980. Jones, Cliff B., *Software Development: A Rigorous Approach*, Prentice-Hall International, London, England, 1980.

Norman, 1986. Norman, Donald A., "Cognitive Engineering," in *User Centered System Design: New Perspectives on Human-Computer Interaction, op.cit.*, pp. 31-61.

Oram, 1989. Oram, Andrew, and Kathleen A. Ferraro, "Sentence First, Verdict Afterward: Finding the Prerequisites for Good Computer Documentation," *Proceedings of the ACM SIGDOC '89 Conference*, ACM, New York, 1989.

Oram, 1991. Oram, Andrew, "The Hidden Effects of Computer Engineering on User Documentation," in *Perspectives on Software Documentation: Inquiries and Innovations*, ed. Thomas T. Barker, Baywood Press, Amityville, NY, 1991.

Probert, 1984. Probert, Robert L., and Hasan Ural, "High-Level Testing and Example-Directed Development of Software Specifications," *Journal of Systems and Software*, vol. 4, pp. 317-325, 1984.

Roman, 1985. Roman, Gruia-Catalin, "A Taxonomy of Current Issues in Requirements Engineering," *IEEE Computer*, vol. 18, no. 4, April 1985, pp. 14-21.

Stoy, 1982. Stoy, J., "Data Types for State of the Art Program Development," in *State of the Art Report: Programming Technology* (Series 10, no. 2), ed. P. J. L. Wallis., Pergamon Infotech Ltd., Maidenhead, Berkshire, England, 1982, pp. 303-320.

Acknowledgments

Among the writers and Quality Assurance staff who have worked on verifiable documentation with me, I particularly want to thank Sara Alfaro-Franco, Kathie Mulkerin, John Morrison, Sharon DeVoie, and Jean Sommer for using and evaluating the method. I am also grateful to MASSCOMP, Concurrent Computer Corporation, and Hitachi Computer Products for supporting the experiments and research effort toward verifiable documentation.

EXPERIENCE WITH THE COST OF DIFFERENT COVERAGE GOALS FOR TESTING

Brian Marick
Motorola, Inc.

In coverage–based testing, coverage conditions are generated from the program text. For example, a branch generates two conditions: that it be taken in the false direction and in the true direction. The proportion of coverage conditions a test suite exercises can be used as an estimate of its quality. Some coverage conditions are impossible to exerise, and some are more cost–effectively eliminated by static analysis. The remainder, the feasible coverage conditions, are the subject of this paper.

What percentage of branch, loop, multi–condition, and weak mutation coverage can be expected from thorough unit testing? Seven units from application programs were tested using an extension of traditional black box testing. Nearly 100% feasible coverage was consistently achieved. Except for weak mutation, the additional cost of reaching 100% feasible coverage required only a few percent of the total time. The high cost for weak mutation was due to the time spent identifying impossible coverage conditions.

Because the incremental cost of coverage is low, it is reasonable to set a unit testing goal of 100% for branch, loop, multi–condition, and a subset of weak mutation coverage. However, reaching that goal after measuring coverage is less important than nearly reaching it with the initial black box test suite. A low initial coverage signals a problem in the testing process.

BIOGRAPHICAL SKETCH

Brian Marick graduated from the University of Illinois in 1981 with a BA in English Literature and a BS in Mathematics and Computer Science. Until 1989, he worked in product development as a programmer, tester, and line manager, while attending graduate school as a hobby. The work reported here was a result of a research project funded by a Motorola Partnerships in Research grant through Motorola's Software Research Division. He is currently applying the techniques and tools reported here to Motorola products, training others in their use, and planning further investigations of cost–effective and predictable testing techniques.

Author's address: Motorola, 1101 E. University, Urbana, IL 61801.
Phone: (217) 384–8500
Email: marick@cs.uiuc.edu or marick@urbana.mcd.mot.com.

1. Introduction

One strategy for testing large systems is to test the low–level components ("units") thoroughly before combining them. The expected benefit is that failures will be found early, when they are cheaper to diagnose and correct, and that the cost of later integration or system testing will be reduced. One way of defining "thoroughly" is through coverage measures. This paper addresses these questions:

(1) What types of coverage should be measured?

(2) How much coverage should be expected from black box unit testing?

(3) What should be done if it is not achieved?

This strategy is expensive; in the last section, I discuss ways of reducing its cost. A different strategy is to test units less thoroughly, or not at all, and put more effort into integration and system testing. The results of this study have little relevance for that strategy, though they may provide some evidence in favor of the first strategy.

Note on terminology: "unit" is often defined differently in different organizations. I define a unit to be a single routine or a small group of closely related routines, such as a main routine and several helper routines. Units are normally less than 100 lines of code.

1.1. Coverage

Coverage is measured by instrumenting a program to determine how thoroughly a test suite exercises it. There are two broad classes of coverage measures. Path–based coverage requires the execution of particular components of the program, such as statements, branches, or complete paths. Fault–based coverage requires that the test suite exercise the program in a way that would reveal likely faults.

1.1.1. Path–based Coverage

Branch coverage requires every branch to be executed in both directions. For this code

```
if (arg > 0)
{
 counter = 0;  /* Reinitialize */
}
```

branch coverage requires that the IF's test evaluate to both TRUE and FALSE.

Many kinds of faults may not be detected by branch coverage. Consider a test–at–the–top loop that is intended to sum up the elements of an array:

```
sum = 0;
while (i > 0)
{
 i -= 1;
 sum = pointer[i];       /* Should be += */
}
```

This program will give the wrong answer for any array longer than one. This is an important class of faults: those that are only revealed when loops are iterated more than once. Branch coverage does not force the detection of these faults, since it merely requires that the loop test be TRUE at least once and FALSE at least once. It does not require that the loop ever be executed more than once. Further, it doesn't require that the loop ever be skipped (that is, that i initially be zero or less). *Loop coverage* [Howden78] [Beizer83] requires these two things.

Multi-condition coverage [Myers79] is an extension to branch coverage. In an expression like

if (A && B)

multi-condition coverage requires that A be TRUE in some test, A be FALSE in some test, B be TRUE in some test, and B be FALSE in some test. Multi-condition coverage is stronger than branch coverage; these two inputs

A == 1, B == 1
A == 0, B == 1

satisfy branch coverage, but do not satisfy multi-condition coverage.

There are other coverage measures, such as dataflow coverage [Rapps85] [Ntafos84]. They are not measured in this experiment, so they are not discussed here.

1.1.2. Fault-based Coverage

In *weak mutation coverage* [Howden82] [Hamlet77], we suppose that a program contains a particular simple kind of fault. One such fault might be using <= instead of the correct < in an expression like this:

if (A <= B)

Given this program, a weak mutation coverage system would produce a message like

"gcc.c", line 488: operator <= might be <

This message would be produced until the program was executed over a test case such that $(A \leq B)$ has a different value than $(A < B)$. That is, we must satisfy a *coverage condition* that

$$(A \leq B) \neq (A < B)$$

or, equivalently, that

$$A = B$$

Notice the similarity of this requirement to the old testing advice: "always check boundary conditions".

Suppose we execute the program and satisfy this coverage condition. In this case, the incorrect program (the one we're executing) will take the wrong branch. Our hope is that this incorrect program state will persist until the output, at which point we'll see it and say "Bug!". Of course, the effect might not persist. However, there's evidence [Marick90] [Offutt91] that over 90% of such faults will be detected by weak mutation coverage. Of course, not all faults are such simple deviations from the correct program; probably a small minority are [Marick90]. However, the hope of weak mutation testing is that a test suite thorough enough to detect such simple faults will also detect more complicated faults; this is called the *coupling effect* [DeMillo78]. The coupling effect has been shown to hold in some special cases [Offutt89], but more experiments are needed to confirm it in general.

There are two kinds of weak mutation coverage. *Operator* coverage is as already described — we require test cases that distinguish operators from other operators. *Operand* coverage is similar — for any operand, such as a variable, we assume that it ought to have been some other one. That is, in

my_function(A)

we require test cases that distinguish A from B (coverage condition $A \neq B$), A from C, A from D, and so on. Since there may be a very large number of possible alternate variables, there are usually a very large number of coverage conditions to satisfy. The hope is that a relatively few test cases will satisfy most of them.

1.2. Feasible Coverage Conditions

All coverage techniques, not just mutation coverage, generate coverage conditions to satisfy. (A branch, for example, generates two conditions: one that the branch must be taken true, and one that it must be taken false.) Ideally, one would like all coverage conditions to be satisfied: 100% coverage. However, a condition may be impossible to satisfy. For example, consider this loop header:

```
for(i = 0; i < 4; i++)
```

Two loop conditions cannot be satisfied: that the loop be taken zero times, and that the loop be taken once.

Not all impossible conditions are this obvious. Programmer "sanity checks" also generate them:

```
phys = logical_to_physical(log);
if (NULL_PDEV == phys)
   fatal_error("Program error:  no mapping.");
```

The programmer's assumption is that every logical device has, at this point, a physical device. If it is correct, the branch can never be taken true. Showing the branch is impossible means independently checking this assumption. There's no infallible procedure for doing that.

In some cases, eliminating a coverage condition may not be worth the cost. Consider this code:

```
if (unlikely_error_condition)
  halt_system();
```

Suppose that **halt_system** is known to work and that the unlikely error condition would be tremendously difficult to generate. In this case, a convincing argument that **unlikely_error_condition** indeed corresponds to the actual error condition would probably be sufficient. Static analysis – a correctness argument based solely on the program text – is enough. But suppose the code looked like this:

```
if (unlikely_error_condition)
  recover_and_continue();
```

Error recovery is notoriously error-prone and often fails the simplest tests. Relying on static analysis would usually not be justified, because such analysis is often incorrect — it makes the same flawed implicit assumptions that the designer did.

Coverage conditions that are possible and worth testing are called *feasible*. Distinguishing these from the others is potentially time-consuming and annoyingly imprecise. A common shortcut is to set goals of around 85% for branch coverage (see, for example, [Su91]) and consider the remaining 15% infeasible by assumption. It is better to examine each branch separately, even if against less deterministic rules.

1.3. Coverage in Context

For the sake of efficiency, testing should be done so that many coverage conditions are satisfied without the tester having to think about them. Since a single black box test can satisfy many

coverage conditions, it is most efficient to write and run black box tests first, then add new tests to achieve coverage.

Test design is actually a two stage process, though the two are usually not described separately. In the first stage, *test conditions* are created. A test condition is a requirement that at least one test case must satisfy. Next, test cases are designed. The goal is to satisfy as many conditions with as few test cases as possible [Myers79]. This is partly a matter of cost, since fewer test cases will (usually) require less time, but more a matter of effectiveness: complex test cases are more "challenging" to the program and are more likely to find faults through chance.

After black box tests are designed, written, and run, a coverage tool reports unsatisfied coverage conditions. Those which are feasible are the test conditions used to generate new test cases, test cases that improve the test suite. Our hope is that there will be few of them.

1.4. The Experiment

Production use of a branch coverage tool [Zang91] gave convincing evidence of the usefulness of coverage. Another tool, GCT, was built to investigate other coverage measures. It was developed at the same time as a variant black box testing technique. To tune the tool and technique before putting them to routine use, they were used on arbitrarily selected code from readily available programs. This early experience was somewhat surprising: high coverage was routinely achieved. A more thorough study, with better record–keeping and a stricter definition of feasibility, was started. Consistently high coverage was again seen, together with a low incremental cost for achieving 100% feasible coverage, except for weak mutation coverage. These results suggest that 100% feasible coverage is a reasonable testing goal for unit testing. Further, the consistency suggests that coverage can be used as a "signal" in the industrial quality control sense [DeVor91]: as a normally constant measure which, when it varies, indicates a potential problem in the testing process.

2. Materials and Methods

2.1. The Programs Tested

Seven units were tested. One was an entire application program. The others were routines or pairs of routines chosen from larger programs in widespread use. They include GNU make, GNU diff, and the RCS revision control system. All are written in C. They are representative of UNIX application programs.

The code tested ranged in size from 30 to 272 lines of code, excluding comments, blank lines, and lines containing only braces. The mean size was 83 lines. Size can also be measured by total number of coverage conditions to be satisfied, which ranged from 176 to 923, mean 479. On average, the different types of coverage made up these percentages of the total:

Branch	Loop	Multi	Operator	Operand
7.5%	3.0%	2.7%	16.1%	70.7%

In the case of the application program, the manual page was used as a specification. In the other cases, there were no specifications, so I wrote them from the code.

The specifications were written in a rigorous format as a list of preconditions (that describe allowable inputs to the program) and postconditions (each of which describes a particular result and the conditions under which it occurs). The format is partially derived from [Perry89]; an example is given in Appendix A.

2.2. The Test Procedure

The programs were tested in five stages. Each stage produced a new test suite that addressed the shortcomings of its predecessors.

2.2.1. Applying a Variant Black Box Technique

The first two test suites were developed using a variant black box testing technique. It is less a new technique than a codification of good practice. Its first stage follows these steps:

Black 1: Test conditions are methodically generated from the *form* of the specification. For example, a precondition of the form "X must be true" generates two test conditions: "X is true" and "X is false", regardless of what X actually is. These test conditions are further refined by processing connectives like AND and OR (using rules similar to cause–effect testing [Myers79]). For example, "X AND Y" generates three test cases:

X is true, Y is true.
X is false, Y is true.
X is true, Y is false.

Black 2: Next, test conditions are generated from the *content* of the specification. Specifications contain *cliches* [Rich90]. A search of a circular list is a typical cliche. Certain data types are also used in a cliched way. For example, the UNIX pathname as a slash–separated string is implicit in many specifications. Cliches are identified by looking at the nouns and verbs in the specification: "search", "list", "pathname".

The implementations of these cliches often contain cliched faults.[1] For example:

(1) If the specification includes a search for an element of a circular list, one test condition is that the list does not include the element. The expectation is that the search might go into an infinite loop.

(2) If a function decomposes and reconstructs UNIX pathnames, one test condition is that it be given a pathname of the form "X//Y", because programmers often fail to remember that two slashes are equivalent to one.

Because experienced testers know cliched faults, they use them when generating tests. However, writing the cliches and faults down in a catalog reduces dependency on experience and memory. Such a catalog has been written; sample entries are given in Appendix B.

Black 3: These test conditions are combined into test cases. A test case is a precise description of particular input values and expected results.

The next stage is called broken box testing[2]. It exposes information that the specification hides from the user, but that the tester needs to know. For example, a user needn't know that a routine uses a hash table, but a tester would want to probe hash table collision handling. There are two steps:

Broken 1: The code is scanned, looking for important operations and types that aren't visible in the specification. Types are often recognized because of comments about the use of a variable. (A variable's declaration does not contain all the information needed; an integer may be a count, a range, a percentage, or an index, each of which produces different test conditions.) Cliched operations are often distinct blocks of code separated by blank lines or comments. Of course, the key

[1] There's some empirical evidence for this: the Yale bug catalogues [Johnson83], [Spohrer85] are collections of such cliched faults, but only those made by novice programmers. More compelling is the anecdotal evidence: talk to programmers, describe cliched errors, and watch them nod their heads in recognition.

[2] The name was coined by Johnny Zweig. This stage is similar to Howden's functional testing: see [Howden80a], [Howden80b], or [Howden87].

way you recognize a cliche is by having seen it often before. Once found, these cliches are then treated exactly as if they had been found in the specification. No attempt is made to find and satisfy coverage conditions. (The name indicates this: the box is broken open enough for us to see gross features, but we don't look at detail.)

Broken 2: These new test conditions are combined into new test cases.

In production use, a tester presented with a specification and a finished program would omit step Black3. Test conditions would be derived from both the specification and the code, then combined together. This would minimize the size of the test suite. For this experiment, the two test suites were kept separate, in order to see what the contribution of looking at the code would be. This also simulates the more desirable case where the tests are designed before the code is written. (Doing so reduces elapsed time, since tests can be written while the code is being written. Further, the act of writing concrete tests often discovers errors in the specification, and it's best to discover those early.)

In this experiment, the separation is artificial. I wrote the specifications for six of the programs, laid them aside for a month, then wrote the test cases, hoping that willful forgetfulness would make the black box testing less informed by the implementation.

2.2.2. Applying Coverage

After the black and broken box tests were run, three coverage test suites were written. At each stage, cumulative coverage from the previous stages was measured, and a test suite that reached 100% feasible coverage on the next coverage goal was written and run. The first suite reached 100% feasible branch and loop coverage, the next 100% feasible multi–condition coverage, and the last 100% feasible weak mutation coverage. The stages are in order of increasing difficulty. There is no point in considering weak mutation coverage while there are still unexecuted branches, since eliminating the branches will eliminate many weak mutation conditions without the tester having to think about them.

In each stage, each coverage condition was first classified. This meant:

(1) For impossible conditions, an argument was made that the condition was indeed impossible. This argument was not written down (which reduces the time required).

(2) Only weak mutation conditions could be considered not worthwhile. The rule (necessarily imprecise) is given in the next section. In addition to the argument for infeasibility, an argument was made that the code was correct as written. (This was always trivial.) The arguments were not written down.

(3) For feasible conditions, a test condition was written down. It described the coverage condition in terms of the unit's input variables.

After all test conditions were collected, they were combined into test cases in the usual way.

2.2.2.1. Feasibility Rules

Weak mutation coverage conditions were never ruled out because of the difficulty of eliminating them, but only when a convincing argument could be made that the required tests would have very little chance of revealing faults. That is, the argument is that the coupling effect will not hold for a condition. Here are three typical cases:

(1) Suppose *array* is an input array and array[0]=0 is the first statement in the program. If array[0] is initially always 0, GCT will complain that the initial and final value of the array are never different. However, a different initial value could never have any effect on the program.

(2) In one program, fopen() always returned the same file pointer. Since the program doesn't manipulate the file pointer, except to pass it to fread() and fclose(), a different file pointer would not detect a fault.

(3) A constant 0 is in a line of code executed by only one test case. In that test case, an earlier loop leaves an index variable with the value 0. GCT complains that the constant might be replaced by the variable. That index variable is completely unused after the loop, it has been left with other values in other tests, and the results of the loop do not affect the execution of the statement in question. Writing a new test that also executes the statement, but with a different value of the index variable, is probably useless.

2.2.2.2. Weak mutation coverage

Weak mutation coverage tools can vary considerably in what they measure. GCT began with the single–token transformations described in [Offutt88] and [Appelbe??], eliminating those that are not applicable to C. New transformations were added to handle C operators, structures, and unions. Space does not permit a full enumeration, but the extensions are straightforward. See [Agrawal89] for another way of applying mutation to C.

Three extensions increased the cost of weak mutation coverage:

(1) In an expression like (*variable* < *expression*), GCT requires more than that *variable*≠*alternate*. It requires that *variable*<*expression* ≠ *alternate*<*expression*. This *weak sufficiency requirement* guards against some cases where weak mutation would fail to find a fault; see [Marick90].

(2) Weak sufficiency also applies to compound operands. For example, when considering the operator **ptr*, GCT requires **ptr*≠**other_ptr*, not just *ptr*≠*other_ptr*. (Note: [Howden82] handles compound operands this way. It's mentioned here because it's not an obvious extension from the transformations given in [Offutt88], especially since it complicates the implementation somewhat.)

(3) Variable operands are required to actually vary; they cannot remain constant.

See [Agrawal89] for another way of applying mutation to C.

2.2.3. What was Measured

For each stage, the following was measured:

(1) The time spent designing tests. This included time spent finding test conditions, ruling out infeasible coverage conditions, deciding that code not worth covering (e.g., potential weak mutation faults) was correct, and designing test cases from test conditions. The time spent actually writing the tests was not measured, since it is dominated by extraneous factors. (Can the program be tested from the command line, or does support code have to be written? Are the inputs easy to provide, like integers, or do they have to be built, like linked lists?)

(2) The number of test conditions and test cases written down.

(3) The percent of coverage, of all types, achieved. This is the percent of total coverage conditions, not just feasible ones.

(4) The number of feasible, impossible, and not worthwhile coverage conditions.

2.2.4. An Example

LC is a 272–line C program that counts lines of code and comments in C programs. It contains 923 coverage conditions.

The manpage was used as the starting specification; 101 test conditions were generated from it. These test conditions could be satisfied by 36 test cases. Deriving the test conditions and designing

the test cases took 2.25 hours. Four faults were found.[3]

These coverages were achieved in black box testing:

	Branch	Loop	Multi	Operator	Operand
Number satisfied	94 of 98	19 of 24	41 of 42	170 of 180	470 of 580
Percent	96%	79%	98%	94%	81%

The next stage was broken box testing. In two hours, 13 more test conditions and 4 more test cases were created. The increase is not large because the implementation of this program is relatively straightforward, with few hidden operations like hash table collision handling. No more faults were found, and the following increases in coverage were seen:

	Branch	Loop	Multi	Operator	Operand
Number newly satisfied	2 of 4	0 of 5	1 of 1	2 of 10	7 of 110
Cumulative Percent	98%	79%	100%	96%	82%

In the next stage, the seven unsatisfied branch and loop conditions required only 15 minutes to examine. Four were impossible to satisfy, two more were impossible to satisfy because of an already–found fault, and one could be satisfied. The resulting test satisfied exactly and only its coverage condition.

Because multi–condition coverage was 100% satisfied, 8 operator test conditions and 103 operand test conditions remained to be satisfied. Of these, 107 were infeasible. 97 of these were impossible (one of them because of a previously–discovered fault), and the remaining 10 were judged not worth satisfying.

Seven new test conditions were written down. Two of these were expected to satisfy the remaining four weak mutation conditions, and the rest were serendipitous. (That is, while examining the code surrounding an unsatisfied condition, I discovered an under–tested aspect of the specification and added tests for it, even though those tests were not necessary for coverage. This is not uncommon; often these tests probe whether special–case code needs to be added to the program. Note that [Glass81] reports that such omitted code is the most important class of fault in fielded systems.)

These seven test conditions led to four test cases. One of the serendipitous test conditions discovered a fault.

Satisfying weak mutation coverage required 3.25 hours, the vast majority of it devoted to ruling out impossible cases.

3. Results

This section presents the uninterpreted data. Interpretations and conclusions are given in the next section.

Measures of effort are given in this table. All measures are mean cumulative percentages; thus weak mutation always measures 100%.

	Black	Broken	Branch+Loop	Multi	Weak
Time	53	74	76	77	100
Test Conditions	79	93	95	95	100
Test Cases	74	89	92	92	100

[3] The program has been in use for some years without detecting these faults. All of them corresponded to error cases, either mistakes in invocation or mishandling of syntactically incorrect C programs.

The next table reports on coverage achieved. Numbers give the percent of total coverage conditions eliminated by testing. An asterisk indicates that all coverage conditions of that type were eliminated (either by testing or because they were infeasible). One number, the 100% for All Path–Based Coverage in the Branch+Loop stage, has a different interpretation. It measures the branches and loops eliminated either by testing or inspection, together with the multi–conditions eliminated by testing alone. This was done because the number should indicate how much remains to be done after the stage.

	Black	Broken	Branch+Loop	Multi	Weak
Branch Coverage	95	99	*	*	*
All Path–based Coverage	92	95	100	*	*
Weak Coverage	84	88	89	89	*

The time spent during the latter stages depends strongly on how many coverage conditions have to be examined. Most were weak mutation conditions: 93% (std. dev. 3%), compared to 5% (std. dev. 3%) loop, 1% (std. dev. 2%) branch, and 0.1% (std. dev. 0.3%) multi–condition.

Because examining an impossible condition is of little use, it is useful to know what proportion of time is spent doing that. This was not measured, but it can be approximated by the proportion of examined conditions which were impossible.

	Branch	Loop	Multi	Weak
Percent Impossible	83	70	0	69
Percent Not Worth Testing	0	0	0	17
Feasible	17	30	100	14

The infeasible weak mutation conditions were the vast majority of the total infeasible conditions (mean 94%, std. dev 5). Of these, 9% (std. dev. 8) were operator conditions, 44% (std. dev. 25) were due solely to the requirement that variables vary, and other operand conditions were 47% (std. dev. 19). The actual significance of the "variables vary" condition is less than the percentage suggests; ruling them out was usually extremely easy.

In any process, consistency of performance is important. This table shows the standard deviations for the first two stages. (The numbers in parentheses are the mean values, repeated for convenience.) For example, during black box testing, 79% of the test conditions were written, with an 18% standard deviation. After broken box testing, the percentage increased to 93% and the standard deviation decreased to 5%. Results for later stages are not given because the mean values are very close to 100% and the variability naturally drops as percentages approach 100%. For the same reason, the apparent decreases shown in this table may not be real. Because the stages are not independent, there seems to be no statistical test that can be applied.

	Black	Broken
Time	19 (53)	14 (74)
Test Conditions	18 (79)	5 (93)
Test Cases	11 (74)	8 (89)
Branch Coverage	6 (95)	3 (99)
All Path–based Coverage	7 (92)	2 (95)
Weak Coverage	11 (84)	4 (88)

The mean absolute time in minutes per line of code is given in this table. The values are cumulative from stage to stage.

	Black	Broken	Branch+Loop	Multi	Weak
Minutes/LOC	2.4	2.8	2.9	2.9	3.6
Std. Deviation	1.4	1.4	1.4	1.4	1.4

4. Discussion

This section first interprets the results, then draws some conclusions about the use of coverage in testing.

Measured by branch coverage alone, the first two stages attained high coverage: 95% for black, 99% for broken[4]. These numbers are higher than those reported in other studies of unit–sized programs. [Vouk86] achieved 88% branch coverage with black–box testing of several implementations of a single specification. [Lauterbach89] found 81% coverage for units selected from production software.

When all of the path–based coverage measures are considered together, the results are similar: 92% for black, 95% for broken. The additional cost to reach 100% on all of the path–based coverage measures was 3% of total time, 2% of total test conditions, and 3% of the total test cases. (If testing had not included weak mutation coverage, the percentages would have been 4% of time, 3% of test conditions, and 4% of test cases.) Coverage testing raised the design cost from 2.8 to 2.9 minutes per line of code. Given that test writing time is proportional to test design time, and that the fixed startup cost of testing is large, this extra expense is insignificant. This result is similar to that of [Weyuker90], who found that stricter dataflow criteria were not much more difficult to satisfy than weaker ones, despite much larger theoretical worst–case bounds.

High path–based coverage is a reasonable expectation. The cost to reach 100% feasible coverage is so low that it seems unwise to let a customer be the first person to exercise a branch direction, a particular loop traversal, or a multi–condition.

The black and broken stages lead to a lower weak mutation coverage: 84% for black and 88% for broken. [DeMillo88] cites an earlier study where black box tests yielded 81% mutation coverage for a single program.

Continuing to branch and loop coverage gained 1%, leaving 11% of the weak mutation conditions. This agrees with [Ntafos84], who found that branch coverage (achieved via random testing) left on average 8% uncovered mutation conditions. (A variant of dataflow testing left 4% uncovered conditions.) It does not agree so well with the study cited in [DeMillo88]. There, branch coverage of a simple triangle classification program left 22% uncovered mutation conditions.[5]

After the multi–condition stage, satisfying the remaining weak mutation conditions is expensive: 8% more test cases and 5% more test conditions, but at the cost of 23% of the total time. The large time is because most coverage conditions (93%) are weak mutation. The relatively low yield is because most of the remaining weak mutation conditions are impossible (69%) or not worth testing (17%). The time spent ruling these out is wasted. GCT could be enhanced to do more of this automatically, but considerable manual work is unavoidable. Further, weak mutation coverage conditions are the hardest to eliminate; the effort is typically greater than, say, forcing a branch to be taken in the TRUE direction. Thus, the cost of weak mutation testing is likely to remain higher.

---

[4] Recall again that I wrote the specifications. The black box numbers are less reliable than the broken box numbers. Recall also that the percentages reported are of total conditions, not just the feasible ones.

[5] Note that these two studies report on strong mutation coverage. However, empirical studies like [Marick90] and [Offutt91] have found that strong mutation coverage is roughly equal to weak mutation coverage.

Of course, those remaining expensive tests might be quite cost effective. They might find many faults. In the near future, GCT and this testing technique will be applied to programs during development. These case studies will determine whether the extra effort of weak mutation coverage is worthwhile by measuring how many faults are detected. In the meantime, weak mutation coverage cannot be recommended.

There is one exception. Relational operator faults ($<$ for $<=$ and the like) are common; indeed, they are the motivation behind testing boundary conditions. As [Myers78] observes, testers often think they are doing a better job testing boundary conditions than they actually are. The relational operator subset of operator weak mutation coverage provides an objective check. Although no records were kept for this subset, few of these conditions remained until the weak mutation stage, and they were generally easy to satisfy. GCT is in the early stages of production use within Motorola, and users are advised to follow the technique described here through branch, loop, multi–condition, and relational operator coverage.

This advice may seem peculiar in one respect. General weak mutation testing is ruled out because it does not seem useful enough for the effort expended. However, broken box testing gains less than 5% coverage but costs 21% of the time, along with 15% of the test cases. It might seem that broken box testing is not worthwhile, but recall how the variability in test conditions and coverage conditions drops sharply from black to broken box testing. This may be an artifact of using percentages. However, it is certainly plausible — broken box testing removes one source of variability, the proportion of internal cliches exposed in the specification. Further, most of the test conditions in broken box testing exist to discover faults of omission: to uncover missing code, rather than to exercise present code. One must remember that coverage, while an important estimator of test suite quality, does not tell the whole story. Like all metrics, it must be used with care, lest the unmeasured aspects of quality be forgotten. The effective use of coverage will be discussed later, after some other important issues are considered.

4.1. Potential Problems with this Study

The coverage in this study is generally higher than that found in other studies. To what extent are these results particular to this technique? Three factors might be important:

(1) The rigorous, stereotyped format of the specification makes it less likely that details needing testing will be lost in the clutter. The same is true of the methodical procedure for deriving test cases. Other formats and procedures should work as well.

(2) The catalog used in black and broken box testing is a collection of the test conditions an expert tester might use, for the benefit of novices and experts with bad memories. It probably contributes to high coverage, but the focus of the catalog is faults of omission — and tests to detect omitted code have no effect on coverage.

(3) Broken box testing brings tester–relevant detail into the specification. It has an effect on coverage (a 3–4% increase in the different coverage measures).

In short, this technique is a codification of existing practice, designed to make that practice more consistent and easier to learn. Other methodical codifications should achieve comparable coverages.

In isolation, this study is too small for firm conclusions. Not enough programs were tested, and they were all tested by one person, who had written the specifications. However, the results are consistent with other experience. In a later experiment, a classful of students applied an extension of this technique to three programs. All the data has not been analysed, but the same pattern appears. The next study will apply the technique to a complete subsystem. It will be used to refine the technique, to try to repeat these results, and to address two additional questions: how well do the different stages detect faults? and what is the effect of differing definitions of feasibility?

4.2. The Effective Use of Coverage

100% feasible coverage appears to be a reasonable goal. How should it be achieved? When coverage is first measured, there will be uncovered conditions. How are they to be handled?

The natural impulse is to add tests specifically designed to increase coverage. This approach, however, is based on a logical fallacy. Because we believe that (1) a good test suite will achieve high coverage, we are also asked to believe that (2) any test suite that achieves high coverage must be good. Yet (1) does not imply (2). In particular, tests designed to satisfy coverage tend to miss faults of missing code, since a tool cannot create coverage conditions for code that ought to be there, but isn't. These are the very faults that [Glass81] found most important in fielded systems.

An unexercised coverage condition should be considered a signal pointing to an under–exercised part of the specification. New test conditions should be derived from that part, not just from the coverage condition that points to it. This may seem an unnecessarily thorough approach, but the results of this study suggest that its cost is only a few percent of the total cost of test design. And, as we saw in the LC example, such "unnecessary" test cases may be the ones that find faults, while the test cases added for coverage do not.

A more important question is this: what is to be done if coverage is substantially lower than 100%?

If high coverage is not achieved, that's not just a signal to improve the test suite, it's also a signal to improve the test ***process***. Suppose the process leads to 60% branch coverage. The remaining coverage conditions can still be used to produce a good test suite, so long as they are treated as pointers into the specification. But it would have been better to produce this good test suite in the first place:

(1) Tests added later are more expensive. Time must be spent understanding why a coverage condition is unexercised.

(2) Tests sometimes discover problems with the specification. It is better to discover those problems before the code is written.

(3) If coverage varies widely from program to program, the cost and duration of testing is less predictable.

Low coverage should lead to a diagnosis of a process problem. Perhaps the test generation technique needs to be improved, or the tester needs better training, or special difficulties in testing this application area need to be addressed. A common problem is that the form or style of the specification obscures or ignores special cases.

4.3. Testing Large Systems

The results of this paper apply only when test cases are derived from individual units. If this same technique is applied to collections of units, or whole systems, the result will be lower coverage. A subsystem or system specification will not contain enough detail to write high–yield test cases.

However, testing each function in isolation is very expensive. In the worst case, special support code ("test harnesses") will have to be written for every function. The cost of support code can dominate the cost of unit testing.

A good compromise is to use subsystems as test harnesses. All routines in a subsystem are tested together, using test conditions derived from their specifications, augmented by test conditions targeted to integration faults. Tests are then added, based on coverage data.

Disadvantages of this approach are:

(1) Extra design effort to discover how to force the subsystem to deliver particular values to the function under test.

(2) Faults found by the tests will be more difficult to isolate.

(3) More coverage conditions will be impossible. For example, the subsystem might not allow the exercising of a function's error cases. Faults in error handling might not be discovered until this "tested" function is reused.

(4) Increased chance that a possible condition will be thought impossible.

Offsetting these is the advantage of not having to write harnesses for all of the functions. The use of the subsystem as the single harness also makes the creation and control of a repeatable test suite easier. This approach should usually be reasonable with subsystems of up to a few thousand lines of code. Raw size is not as important as these factors:

(1) Clean boundaries between the subsystem and other subsystems. This reduces the cost of building a harness for the subsystem.

(2) Some support for testing built into the subsystem. Relatively simple changes can often make testing much easier. Because the changes are simple, they can often be retrofitted.

(3) A single developer responsible for the subsystem. This reduces the cost of test design and sharply reduces the cost of diagnosis.

Once coverage has been achieved at the subsystem level, it need not be repeated at large–scale integration testing or system testing. That testing should be targeted at particular risks, preferably identified during system analysis and design. 100% coverage is not relevant.

This strategy, "unit–heavy", is probably not often done. Usually more effort is devoted to integration and (especially) system testing. Sometimes unit testing is omitted entirely. Other times it is done without building test harnesses or repeatable test suites, which essentially means omitting unit testing during maintenance. The reason for this "unit–light" strategy is a risk/benefit tradeoff: the extra cost of discovering faults later is expected to be less than the cost of more thorough early testing.

What relevance has this paper to that strategy? The results do not apply. However, it may provide more evidence that the unit–heavy strategy is reasonable:

(1) It provides a measurable stopping criterion, 100% feasible coverage, for testing. Testers often suffer from not having concrete goals.

(2) The criterion is intuitively reasonable. It makes sense to exercise the branches, loops, and multi–conditions, and to rule out off–by–one errors.

(3) As an objective measure, the criterion can be used to achieve more consistent and predictable testing.

(4) The cost can be reduced by using subsystem harnesses, and can be greatly reduced if testing is considered during design.

However, there is no hard data on which to base a choice. Further studies are being planned to get a clearer idea of the relative costs and benefits of the two strategies (which, of course, form a continuum rather than two distinct points). In the meantime, programmers, testers, and managers can examine bug reports from the field and ask

(1) Would 100% coverage have forced the early detection of this fault?

(2) Would thorough black box tests have forced the detection of the fault? (A somewhat more subjective measure.)

[Howden80a] took this approach with 98 faults discovered after release of edition five of the IMSL library. He found that black box testing would have found 20% of the faults, applying black box techniques to internals would have found 18%, and branch coverage would have forced 13%. (Some faults would be found by more than one technique.) There is danger in extrapolating these numbers to other systems: they are for library routines, they depend on how the product was tested before release, and they do not describe the severity of the faults. There is no substitute for

evaluating your own testing process on your own products.

Appendix A: An example specification

This is a part of the specification for the compare_files() routine in GNU diff. The actual specification explains the context.

COMPARE_FILES(DIR1, FILE1, DIR2, FILE2, DEPTH)
All arguments are strings, except DEPTH, which is an integer.

PRECONDITIONS:
1. Assumed: At most one of FILE1 and FILE2 is null.

2. Assumed: If neither of FILE1 and FILE2 is null
THEN they are string–equal.

[...]
4. Validated: if FILE1 is non–NULL, then file DIR1/FILE1 can be
opened for reading
On failure:
An error message is printed to standard error.
The return value is 2.

[...]

POSTCONDITIONS:
1 IF FILE1 and FILE2 are plain files that differ
THEN the output is a normal diff:

```
1c1
< bar
---
> foo
```

and compare_files returns 1.

2. If FILE1 and FILE2 are identical plain files
THEN there is no output and the return value is 0.

3. IF FILE1 is a directory, but FILE2 is not
THEN
the output is "FILE1 is a directory but FILE2 is not"
the return value is 1.

[...]

Appendix B: Example of catalog entries

This appendix shows the catalog entries that apply to the examples given in the text.

23. GENERAL SEARCHING CONDITIONS

• Match not found

- Match found (exactly one match in collection)
- More than one match in the collection
- Single match found in first position (it's not the only element) DEFER
- Single match found in last position (it's not the only element) DEFER

Note that these conditions apply whether the search is forward or backward.

There is more detail to the technique than explained in this paper, aimed toward reducing the amount of work; the DEFER keyword is part of that detail.

19. PATHNAMES

...

19.1. Decomposing Pathnames

There are many opportunities for errors when decomposing pathnames into their component parts and putting them back together again (for example, to add a new directory component, or to expand wildcards).

- <text>/
- <text>/<text>
- <text>/<text>/<text>
- <text>//<text>

Also consider the directory and file components as Containers of variable–sized contents.

REFERENCES

[Agrawal89]
H. Agrawal, R. DeMillo, R. Hathaway, Wm. Hsu, Wynne Hsu, E. Krauser, R.J. Martin, A. Mathur, and E. Spafford. *Design of Mutant Operators for the C Programming Language.* Software Engineering Research Center report SERC–TR–41–P, Purdue University, 1989.

[Appelbe??]
W.F. Appelbe, R.A. DeMillo, D.S. Guindi, K.N. King, and W.M. McCracken, *Using Mutation Analysis For Testing Ada Programs.* Software Engineering Research Center report SERC–TR–9–P, Purdue University.

[Beizer83]
Boris Beizer. *Software Testing Techniques.* New York: Van Nostrand Reinhold, 1983.

[DeMillo78]
R.A. Demillo, R.J. Lipton, and F.G. Sayward, "Hints on test data selection: help for the practicing programmer". *Computer.* vol. 11, no. 4, pp. 34–41, April, 1978.

[DeMillo88]
R.A. DeMillo and A.J. Offutt. "Experimental Results of Automatically Generated Adequate Test Sets". *Proceedings of the 6th Annual Pacific Northwest Software Quality Conference,*

Portland, Oregon, September 1988.

[DeVor91]
R.E. DeVor, T.H. Chang, and J.W. Sutherland. *Statistical Methods for Quality Design and Control.* New York: MacMillan, 1991 (in press).

[Glass81]
Robert L. Glass. "Persistent Software Errors". *Transactions on Software Engineering*, vol. SE–7, No. 2, pp. 162–168, March, 1981.

[Hamlet77]
R.G. Hamlet. "Testing Programs with the aid of a compiler". *IEEE Transactions on Software Engineering*, vol. SE–3, No. 4, pp. 279–289, 1977.

[Howden78]
W. E. Howden. "An Evaluation of the Effectiveness of Symbolic Testing". *Software – Practice and Experience*, vol. 8, no. 4, pp. 381–398, July–August, 1978.

[Howden80a]
William Howden. "Applicability of Software Validation Techniques to Scientific Programs". *Transactions on Programming Languages and Systems*, vol. 2, No. 3, pp. 307–320, July, 1980.

[Howden80b]
W. E. Howden. "Functional Program Testing". *IEEE Transactions on Software Engineering*, vol. SE–6, No. 2, pp. 162–169, March, 1980.

[Howden82]
W. E. Howden. "Weak Mutation Testing and Completeness of Test Sets". *IEEE Transactions on Software Engineering*, vol. SE–8, No. 4, pp. 371–379, July, 1982.

[Howden87]
W.E. Howden. *Functional Program Testing and Analysis.* New York: McGraw–Hill, 1987.

[Johnson83]
W.L Johnson, E. Soloway, B. Cutler, and S.W. Draper. *Bug Catalogue: I.* Yale University Technical Report, October, 1983.

[Lauterbach89]
L. Lauterbach and W. Randall. "Experimental Evaluation of Six Test Techniques". *Proceedings of COMPASS 89*, Washington, DC, June 1988, pp. 36–41.

[Marick90]
B. Marick. *Two Experiments in Software Testing.* Technical Report UIUCDCS–R–90–1644, University of Illinois, 1990. Portions are also to appear in *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification.*

[Myers78]
Glenford J. Myers. "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections". *Communications of the ACM*, Vol. 21, No. 9, pp. 760–768, September, 1978.

[Myers79]
Glenford J. Myers. *The Art of Software Testing.* New York: John Wiley and Sons, 1979.

[Ntafos84]
Simeon Ntafos. "An Evaluation of Required Element Testing Strategies". *Proceedings of the 7th International Conference on Software Engineering*, pp. 250–256, IEEE Press, 1984.

[Offutt88]
A.J. Offutt. *Automatic Test Data Generation.* Ph.D. dissertation, Department of Information and Computer Science, Georgia Institute of Technology, 1988.

[Offutt89]
A.J. Offutt. "The Coupling Effect: Fact or Fiction". *Proceedings of the ACM SIGSOFT 89 Third Symposium on Software Testing, Analysis, and Verification*, in *Software Engineering Notes*, Vol. 14, No. 8, December, 1989.

[Offutt91]
A.J. Offutt and S.D. Lee, *How Strong is Weak Mutation?*, Technical Report 91–105, Clemson University Department of Computer Science, 1991. Also to appear in *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification.*

[Perry89]
Dewayne E. Perry. "The Inscape Environment". *Proceedings of the 11th International Conference on Software Engineering*, pp. 2–12, IEEE Press, 1989.

[Rapps85]
Sandra Rapps and Elaine J. Weyuker. "Selecting Software Test Data Using Data Flow Information". *Transactions on Software Engineering*, vol. SE–11, No. 4, pp. 367–375, April, 1985.

[Rich90]
C. Rich and R. Waters. *The Programmer's Apprentice.* New York: ACM Press, 1990.

[Spohrer85]
J.C. Spohrer, E. Pope, M. Lipman, W. Scak, S. Freiman, D. Littman, L. Johnson, E. Soloway. *Bug Catalogue: II, III, IV.* Yale University Technical Report YALEU/CSD/RR#386, May 1985.

[Su91]
Jason Su and Paul R. Ritter. "Experience in Testing the Motif Interface". *IEEE Software*, March, 1991.

[Vouk86]
Mladen A. Vouk, David F. McAllister, and K.C. Tai. "An Experimental Evaluation of the Effectiveness of Random Testing of Fault–Tolerant Software". In *Proceedings of the Workshop on Software Testing Conference*, pp. 74–81, Banff, Canada, 1986.

[Weyuker90]
Elaine J. Weyuker. "The Cost of Data Flow Testing: An Empirical Study". *Transactions on Software Engineering*, vol. SE–16, No. 2, pp. 121–128, February, 1990.

[Zang91]
Xiaolin Zang and Dean Thompson. "Public–Domain Tool Makes Testing More Meaningful" (review). *IEEE Software*, July, 1991.

On the Relative Strengths of Data Flow and Mutation Based Test Adequacy Criteria

Aditya P. Mathur

Software Engineering Research Center and Department of Computer Science
Purdue University
W. Lafayette, IN 47907

July 18, 1991

Abstract

Data flow and mutation testing have existed for a long time as two powerful clear-box testing techniques. Each technique provides a mechanism for measuring the adequacy of the test data used for testing a program. There has been considerable ongoing debate in the testing community regarding the "power" of the two techniques. Adequate theories of mutation and data flow testing do not exist to prove that one method is more powerful than the other. We are also not aware of any attempt in the past to compare the two techniques. We therefore decided to compare them experimentally. Towards this end, we conducted experiments to find which of these two adequacy criteria is the *stronger* one. This paper presents the experimental methodology used, the empirical data obtained, and our conclusions based on statistical analysis of this data. We also compare the cost of data flow and mutation using the number of test cases to obtain 100% coverage as the cost measure. Data obtained from our experiments, conducted on a set of 18 programs in which the number of decisions vary from 2 to 28, provide strong evidence in favor of mutation.

Index Terms- Data flow testing, mutation testing, test adequacy criteria.

1 Introduction

A variety of clear-box testing techniques are available to a program tester. It is therefore natural to ask the question: *Which of these techniques is the best*? This paper reports an empirical study that was conducted to answer this question for data flow [6, 18] and mutation testing [8]. Even though there has been an informal debate amongst researchers on the effectiveness of these two techniques, there remains a lack of experimental studies providing a comparison. Budd's [4] dissertation does provide data that tends to support the hypothesis that mutation testing is more powerful than data flow testing. However, as discussed in Section 8 the evidence is indirect and inconclusive.

In the past, researchers have examined the effectiveness of various testing techniques [4, 7, 11]. However, to our knowledge a formal comparison of data flow and mutation has not been attempted, perhaps due to a lack of automated tools that support these two techniques. Recently the availability of Mothra [5] and ASSET [15] has made it possible to experiment with mutation and data flow.

The remainder of this paper is organized as follows. Section 2 presents an overview of data flow and mutation techniques. The hypothesis tested to conclude which of the two criteria is stronger is formulated in Section 3. Section 4 outlines the experimental methodology used in our study. Data obtained from the experiments appear in Section 5. Analysis of the data appears in Section 6. Possible pitfalls of our experimental methodology and attempts to avoid them are described in Section 7 We conclude with some remarks on our study and other planned studies in Section 8. An appendix summarizes the terminology used in this paper.

2 An Overview of Data Flow and Mutation Testing

We present a brief overview of data flow and mutation testing. Clarke *et al.* [6] and DeMillo *et al.* [8] provide a detailed description of these two techniques. In our discussion below, P denotes the program under test, T a set of zero or more test cases on which P is executed during testing, and t an element of T. We add subscripts and superscripts to P and T to indicate a specific program or test set. $\mid T \mid$ denotes number of test cases in T. $\mathcal{D}$ denotes the set of all possible subsets of test cases that are in the input domain of P. For example, if P is the program to find the GCD of two integers, then $\mathcal{D}$ is the set of all subsets of pairs of integers where each integer is in the range accepted by the machine on which P is being tested.

Data flow and mutation testing are used for determining the *adequacy* of T. $P(t)$ denotes the output obtained by executing P on t. We also assume that for each test case $t \in T$, $P(t)$ is correct. Thus there exists an oracle that can determine if $P(t)$ is correct or not w.r.t. the specifications from which P is derived.

2.1 Data flow testing

For definitions of ALL-DU paths we follow Clarke *et al.* [6]. A statement in P that can assign a value to a variable x is considered a definition of x. We denote such a statement by S_d^x. For example, an assignment or an input statement can define a variable. The appearance of x in an expression within a statement is considered a *use* of x. We denote such a statement by S_u^x. A use of variable x in an expression that occurs in a conditional branch statement, is said to be a *p-use* of x, otherwise it is called a *c-use*. Let p_x denote a path from S_d^x to some statement in the program that can be executed immediately after S_u^x and that there is no definition of x along this path except at S_d^x. p_x is known as a DU-path. If there is a loop along p_x, then it is sufficient to consider only zero or a single traversal of the loop. Notice that each traversal of the loop results in a unique p_x.

T is said to satisfy the ALL-DU paths criteria if the execution of P on all elements of T causes each p_x for every variable x in P to be traversed at least once. There are several other data flow criteria proposed in [6, 14]. However, it has been shown in [6] that the ALL-DU paths criteria subsumes all other criteria. Thus in this paper we consider only the ALL-DU paths criteria. Note that the ALL-paths criteria subsumes the ALL-DU paths criteria [6]. However, testing for all paths is impractical for anything but small programs due to an arbitrarily large number paths generated by the presence of loops in P.

A path p_x is considered *feasible* if there exists at least one test case t such that executing P on t forces the traversal of p_x. Tools such as ASSET and ATAC [10] find various paths p_x in a program by constructing some form of a flow graph of P. The flow graph can be used to compute all p_x for each variable x. However, not every p_x may be *feasible*. This leads to a modification of our definition of the ALL-DU paths criteria. We say that a test set T *weakly* satisfies the ALL-DU paths criteria if the execution of P on all elements of T causes each of the feasible paths p_x to be traversed at least once [18]. We denote the total number of feasible DU-paths in a program by $\mid DU \mid$.

Definition 1 *A test set is considered adequate w.r.t. data flow testing if it weakly satisfies the ALL-DU paths criteria. Such a test set is denoted by T^d. The ratio of the number of DU-paths covered to the total number of feasible DU-paths is the* **data flow score** *of a test set T w.r.t. the program under test. We refer to this score as F_T.*[1]

2.2 Mutation testing

Mutation testing, also referred to as mutation analysis, is a fault based technique. It provides a set O of operators, also known as *mutant* operators, that model one or more of the faults in a set F of faults. For example, the use of an incorrect variable in an expression is a fault. Mutant operator *svr* in Table 1 models this fault. There is no uniqueness to O and F. In our experiments we used the set of mutant operators available in Mothra and listed in Table 1. More details regarding the design of O and the determination of F may be found in [1].

A mutant operator is *applied* to the program P under test. Such an application transforms P into a similar, though a different, program known as a *mutant*. One application of a mutant operator may generate zero or more mutants. If P contains several entities that are in the domain of a mutant operator, then the operator is applied to each such entity, one at a time. Each application generates a distinct mutant. As an example, consider the mutant operator *sdl* that deletes a statement from P. All statements in P are in the domain of this operator. When applied, *sdl* generates as many mutants as there are statements in P. Each mutant will be identical to P except that it does not contain the single deleted statement. Such mutants can be considered as the *fault induced* versions of P.

Certain mutants are *instrumented* versions of P. Almost any testing tool that measures coverage, such as special value coverage or statement coverage, would also generate such "mutants" which are essentially instrumented versions, or more likely one instrumented version, of the program under test. To be consistent in its naming convention, the mutation tool that we use, namely Mothra, calls these instrumented versions as *mutants* .

The instrumentation is designed to reveal some kind of coverage. For example, a mutant operator that provides special value coverage for variables generates one mutant for each occurrence of a scalar reference. When executed this mutant informs the tester whether or not the desired

[1] We use symbols F and U while referring to data Flow and mUtation scores to avoid conflicts with symbols M and D that are used in other context.

Table 1: Mutant operators used by Mothra.

O[§]	Meaning	Example[†]
aar	array reference for array reference replacement	A(I)=B(J)+3 → A(I)=A(I)+3
abs	special value coverage	P=X*Y+1 → P=zpush(X)*Y+1
acr	array reference for constant replacement	FOUND=0 → FOUND=A(I)
aor	arithmetic operator replacement	A=B+C → A=B-C
asr	array reference for scalar variable replacement	P=X * A(I) → P=X*A(A(I))
car	constant for array reference replacement	P=X * A(I) → P=X * 0
cnr	comparable array name replacement	A(I)=B(J)+3 → A(I)=A(J)+3
crp	constant replacement	do 10 I=1, N → do 10 I=1, N, 2
csr	constant for scalar replacement	P=X * A(I) → P=X * A(1)
der	DO statement end replacement	do 10 I=1, N→ ONETRIP 10 I=1, N
dsa	data statement alterations	DATA X/0/ → DATA X/1/
glr	goto label replacement	goto 20 → goto 10
lcr	logical connector replacement	X .AND. Y → X .OR. Y
ror	relational operator replacement	X .EQ. A(I) → X .GE. A(I)
rsr	return statement replacement	P = X * A(I) → RETURN
san	statement analysis	Each statement replaced by TRAP.
sar	scalar variable for array reference replacement	P=X * A(I) → P=X * Y
scr	scalar for constant replacement	P = 0 → P = X
sdl	statement deletion	Each statement is deleted.
svr[‡]	scalar variable replacement	P = X * A(I) → P = Y * A(I)
uoi	unary operator insertion	P=Q+R → P=Q+R+1

[§] A mutant operator is usually referred to by this mnemonic.

[†] $x \rightarrow y$ means that string x in P is replaced by string y to obtain a mutant. $zpush(x)$ returns x if $x \neq 0$ else it terminates program execution. TRAP causes program termination. ONETRIP terminates program execution after one execution of the do loop body.

[‡] Any variable or constant replacement is carried out w.r.t. variable names and constants already used in the program. For example, the svr mutant operator will replace a variable name X by another name Y only if Y occurs somewhere in the program being mutated.

special value (e.g. 0) was attained by the variable in a specific context, e.g., within an expression in a statement.

Once generated, a mutant M is executed against elements of T. For a fault induced mutant, for any $t \in T$, if the output of M differs from that of P, we say that M is *killed*. An instrumented mutant is killed when a trap function, such as *zpush* or *onetrip*, inserted into P by the mutant operator, terminates mutant execution.[2] A mutant not killed by any $t \in T$ is considered to be *live*. A live mutant M implies that either (a) T needs to be augmented with additional test cases that can kill M or (b) M is *equivalent* to P. A mutant M is considered *equivalent* to P if for all test cases in the input domain of P, the outputs of M and P are identical. Formally, M is equivalent to P if $\forall T \in \mathcal{D}$, $\forall t \in T$, $P(t) = M(t)$.

Definition 2 *A test set is considered adequate w.r.t. mutation testing criteria if it kills all non-equivalent mutants of P. Such a test set is denoted by T^m. The ratio of the number of mutants killed to the number of non-equivalent mutants generated is the* **mutation score** *of a test set T w.r.t. to P. We shall refer to this score as U_T.*

A tool based on mutation analysis, such as Mothra [5], automates several of the tasks implicit in the above description. For example, Mothra performs the tasks associated with mutant generation, mutant execution, live/kill analysis, test case management, and automatic test case generation.

3 Formalizing the "Strength" of an Adequacy Criteria

The primary goal of our experiment was to compare the data flow and mutation test adequacy criteria defined above. To be able to perform such a comparison, we begin by formalizing the meaning of "stronger adequacy criteria". Let T^A denote the test data that satisfies test adequacy criteria A. Let B_{T^A} denote the score computed using criteria B by executing P on T^A. Thus, for example, if B denotes the mutation testing criteria and A the data flow criteria then B_{T^A} denotes the mutation score obtained by executing P on data flow adequate test set. We use F and U to denote, respectively, the data flow and mutation criteria. The following definition characterizes an ideal comparison.

Definition 3 *Given two test adequacy criteria A and B, we say that A is stronger than B if for every program P and $\forall T \in \mathcal{D}$, $B_{T^A} > A_{T^B}$.*

The above definition is at best impractical. To be able to compare mutation and data flow using the above definition, we need a sound theory of mutation which does not exist at present. One way to obtain a practical definition is to use the mean scores obtained by conducting an experiment on finite size sample of programs. Further, for non-trivial programs, it may be impossible to obtain all possible adequate test sets as implied by the above definition. Hence we restrict ourselves to one adequate test set for one program and one adequacy criteria. We then use non-parametric statistical tests to conclude about about the entire population of programs and tests. Towards this end, let $\bar{B}_{T^A}$ ($\bar{A}_{T^B}$) denote the mean score obtained using criteria B (A) by executing a set of programs on data that is found adequate w.r.t. criterion A (B). We now formulate the following null and alternate hypotheses for criteria A and B:

[2] When executed, a trap function merely indicates whether some condition is true or not. Such an indication is used to determine if a coverage criterion has been satisfied. For example, when the function *zpush(x+y)* is executed and the condition $x + y = 0$ is true then the special value coverage criterion, that the expression attain the value 0, is satisfied for the expression $x + y$.

$$H_0: \quad \bar{B}_{TA} = \bar{A}_{TB} \tag{1}$$

$$H_1: \quad \bar{B}_{TA} > \bar{A}_{TB} \tag{2}$$

We use the following definition of *statistically stronger* criteria which can be used in practice.

Definition 4 *Given two test adequacy criteria A and B, we say that A is statistically stronger than B if the null hypothesis as stated above can be rejected.*

The above definition assumes that statistically sound experiments are conducted to test H_0. In the remainder of this paper we use the above definition of statistically stronger criteria to compare data flow and mutation. Below, we loosely refer to "statistically stronger" simply as "stronger".

4 Experimental Methodology

Our experiments were designed to test if H_0 could be rejected when A is the mutation criteria and B is the ALL-DU paths criteria. We conducted two independent sets of experiments referred to as SET-I and SET-II. Each set provided data to test the null hypothesis. Further, data from each set also served as a check against data from the other.

Group composition: Each of the two sets of experiments was conducted by one group of graduate students[3]. The groups that conducted SET-I and SET-II consisted of 3 and 4, members respectively. Group formation was voluntary on the part of students. An arbitrary limit of 4 was placed on the size of each group.

Program selection: Each group selected a set of programs to work with. We must confess that the choice of programs was arbitrary to some extent. However, it was guided by the following factors:

1. For each program selected, two versions were required. One was a Pascal version for use with ASSET and the other a Fortran version for use with Mothra. As we did not have any such programs readily available, we selected some Fortran (Pascal) programs and translated them to Pascal (Fortran).

2. We needed a "large enough" sample of mutation and data flow scores for each program to be able to test the null hypothesis.

3. Mutation and data flow scores of 1 had to be obtained for each program selected. The expense associated with this task depends, largely, on the number of DU-paths and the number of mutants generated for each program.

The above factors led to the selection of 14 programs for SET-I and 9 for SET-II. 5 of the 14 programs were common to both sets. We do not see any reason that this commonality will affect the validity of our conclusions in any way. Of the 18 distinct programs selected, 8 were originally in Pascal and the remaining 10 in Fortran. The set of Pascal programs we used is a subset of the programs used by Weyuker in an empirical study[18]. 8 of the Fortran programs were from a test suite that has been used by Offutt[13] and 2 were ACM algorithms #201 (shellsort) and #202 (perle). All Pascal programs were taken from[12]. In all, 4 programs were in the area of string manipulation, 1 in numerical computing, 3 in sorting/searching, and the remaining in other categories.

Experiments: Each group performed the experiments as shown in Figure 1. Following is a description of each step shown in Figure 1.

[3]The entire project was a part of a graduate Software Engineering course. However, some seniors in the course also participated in the experiments reported.

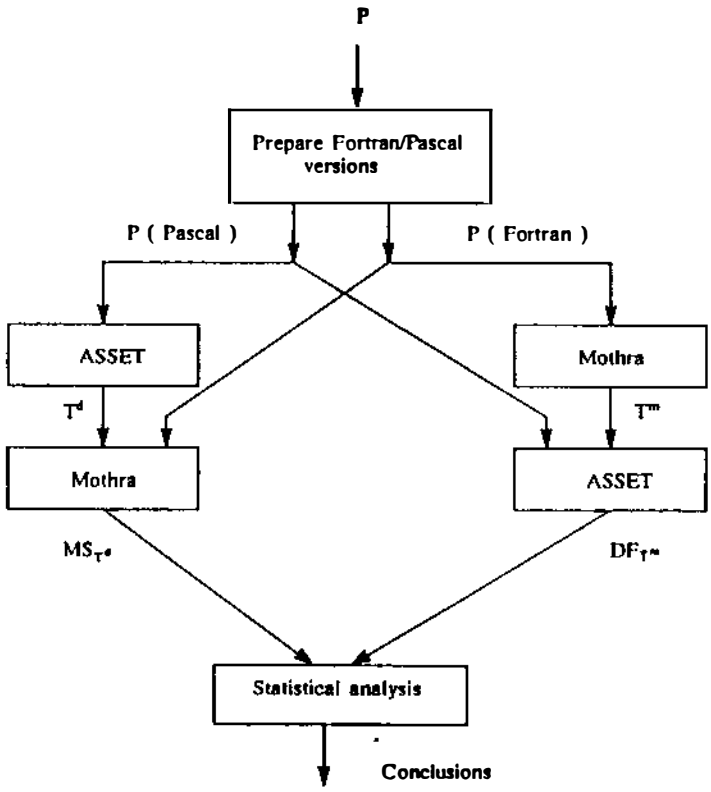


Figure 1: Generating data to compare the strengths of data flow and mutation adequacy criteria.

1. *Program preparation*: Each group began by preparing a Fortran and a Pascal version of each program. Preparation involved rewriting the program and testing it to make sure that both the Fortran and Pascal versions provided identical results. Only functional testing was used in this step.

2. *Test data generation*: For each program test set T^d was constructed using ASSET. Another test set T^m was constructed using Mothra. Note that ASSET or Mothra help in determining the data flow or mutation score of a test set. This score, and the information on what is not covered or which mutants are still live, helps the tester in constructing a new test case. This test case is constructed with the aim of improving the coverage or killing one or more mutants. If the test case succeeds in doing so, it is added to the existing test set, otherwise it is ignored and another test case constructed.

3. *Cross scoring*: Mothra was used to compute the mutation score U_{T^d} for each program. Similarly ASSET was used to compute the data flow score F_{T^m} for each program. We shall refer to these scores as *cross scores.*

The generation of T^d for a program P began by starting with an arbitrary test case t_1 and measuring the DU-path coverage of P. The paths that were not covered by t_1 were examined and another test case constructed to cover more paths. This process was continued until the desired score was obtained. This process obviously required the determination of whether a path is indeed feasible or not[4].

[4]Instructions were given to the experimenters to try their best to "prove" that indeed a path was infeasible. However, there was no independent check of such "proofs". The same is true for the determination of equivalent

Table 2: Some statistics on programs used in the experimentation.

Program	$\mid DU \mid$	$\mid M \mid$	DE
archive	12	547	10
bubble	35	330	3
case	6	113	4
entab	33	775	5
expand	10	643	6
find	72	983	7
gauss	185	3618	20
gauss-main	15	147	3
getfns	43	1300	6
gtext	25	467	5
makepat	134	3899	19
max	19	120	2
omatch	93	1997	28
pattern	22	394	5
perle	60	933	8
shellsort	27	549	4
triangle	135	951	17
unrotate	43	2561	7
averages	62	887	7.33

The generation of T^m also began with an arbitrary test case t_1 and measuring its mutation score on P. Mutants not killed by t_1 were examined individually and additional test cases generated to kill any remaining mutants. The generation process terminated when the only live mutants were the ones equivalent to P.

Cross scoring was relatively easy. It involved executing program P on each element of T^d using Mothra to obtain U_{T^d} and also executing it on each element of T^m to obtain F_{T^m} using ASSET. Data collected during these experiments was then subjected to statistical analysis.

5 Experimental Results

In this section, we present the data collected from the experiments described above. For each program used by the two groups, Table 2 lists the total number of DU-paths, denoted by $\mid DU \mid$, number of mutants generated by applying all the operators listed in Table 1, denoted by $\mid M \mid$, and the number of conditions in the program, denoted by DE. Each `DO` or `for` statement was counted as one condition. Each simple condition inside a compound condition was counted as one condition. For example, the compound condition $a < b \wedge c > d$ contains two simple conditions $a < b$ and $c > d$.

From Table 2 we note that *gauss* generates the largest number of DU-paths (185) and *makepat* generates the largest number of mutants (3899). The smallest number of DU-paths is in *case* (6). The smallest number of mutants (113) is also generated from *case*. Program *max* contains the

mutants. However, data generated by the two independent groups was checked for any major inconsistencies such as the differences between the number of test cases generated for the programs common to the two sets.

Table 3: Results of experimental SET-I.

Program	$\mid T^m \mid$	F_{T^m}	$\mid T^d \mid$	U_{T^d}
archive	11	1	7	0.89
case	6	1	4	0.96
entab	5	1	9	0.95
expand	11	1	4	0.95
find	9	0.94	9	0.94
gauss-main	6	1	6	0.93
getfns*	10	0.88	5	0.98
gtext*	6	0.90	3	0.96
makepat	30	0.96	19	0.94
max	9	1	9	1
omatch	40	1	19	0.92
pattern	8	1	3	0.88
triangle	53	1	10	0.8
unrotate	8	0.92	6	0.9
averages	15.1	0.97	8.0	0.93

* For these programs, $F_{T^m} - U_{T^d} < 0$. See text in Section 6.1

smallest number of conditions (2) and *gauss* the largest (20). Notice that the variability in the number of DU paths, mutants and decisions is an order of magnitude.

Tables 3 and 4 contain the results of SET-I and SET-II, respectively.

6 Analysis of Experimental Data

In this section we first examine the data presented in the previous section. Then we present the results of some statistical procedures applied to this data. We use statistical procedures to compare the strengths of the data flow and mutation criteria and also to compare the costs of these two

Table 4: Results of experimental SET-II.

Program	$\mid T^m \mid$	F_{T^m}	$\mid T^d \mid$	U_{T^d}
bubble	11	1	4	0.87
case	6	1	4	0.98
find	21	0.96	10	0.93
gauss	27	0.99	9	0.90
max	8	1	5	0.93
pattern	15	1	2	0.92
perle	17	1	7	0.95
shellsort	4	0.92	4	0.86
triangle	39	1	10	0.81
averages	16.4	0.98	6.1	0.9

criteria as defined by Weyuker [18].

From Tables 3 and Table 4 we notice that data flow scores obtained using mutation adequate data and mutation scores obtained using data flow adequate data are consistently above or at 0.8. However, to compare the two, let us examine the data in more detail. From Table 3 we find that for 9 out of 14 programs (64%) studied we could obtain a perfect data flow score of 1 using mutation adequate test data. On the contrary we could obtain a mutation score of 1 using data flow adequate test data for only 1 out of 14 programs (7%). From Table 4 we notice that for 6 out of 9 programs (66%) we could obtain a perfect data flow score of 1 using mutation adequate test data. We could not obtain a mutation score of 1 for any program using data flow adequate test data. We also note that in SET-I there are only 2 programs, *getfns* and *gtext*, for which the data flow score from mutation adequate data was lower than the mutation score from data flow adequate data. No such case was encountered in SET-II.

From the above simple analysis it appears that indeed mutation is a stronger adequacy criteria than data flow because data flow score from mutation adequate data is, in most cases, higher than the mutation score from data flow adequate data. However, we wanted to know if the difference in the means of the two samples corresponding to F_{T^m} and U_{T^d} are statistically different. Both the Chi-square test and the Kolmogorov-Smirnov tests failed on the two samples to show that the samples are from a normal distribution. We therefore decided to use non-parametric methods [3] to test for the null hypothesis stated in (1).

6.1 Testing the null hypothesis

Notice that for each program the scores F_{T^m} and U_{T^d} can be treated as a pair. In SET-I and SET-II, we have 14 and 9 such pairs, respectively. We therefore decided to use the sign test and the Wilcoxon signed-rank test to test the stated hypothesis.

To apply the sign test, let S denote the number of differences with positive signs. From Table 3 we find that 12 out of a total of 14 differences ($F_{T^m} - U_{T^d}$) are positive. From the binomial table[5] we find that $Pr[S \geq 10] = 0.029$. This implies that the null hypothesis can be rejected at a significance level (α) of 0.029. From the same table we also find that $Pr[S \geq 11] = 0.006$. This implies that the null hypothesis can also be rejected at $\alpha = 0.006$. For SET-II we obtain $S = 9$. Thus, in this case the null hypothesis can be rejected at any significance level.

To apply the Wilcoxon signed-rank test we computed the W^+ statistic to be 85 and 45 for SET-I and SET-II, respectively. Using the tail probabilities for the null distribution of Wilcoxon's signed-rank statistic we find that $Pr[W^+ \geq 84] = 0.029$. This implies that the null hypothesis can be rejected at $\alpha = 0.029$. For SET-II, we find $Pr[W^+ \geq 42] = .01$. This implies that the null hypothesis can be rejected at $\alpha = .01$.

From the above analysis we find that the null hypothesis is rejected in all cases if we set $\alpha = 0.05$. If we set $\alpha = 0.01$ then the sign test results in the rejection of the null hypothesis for both cases and the singed-rank test rejects the null hypothesis only for SET-II.

6.2 Cost of the two criteria

In [18], Weyuker reported experiments revealing that the cost of the data flow criteria as measured in terms of the number of test cases, is linear in the number of decisions in the program. No such results are available for mutation. Using our experimental data we computed the relationship between the size of mutation adequate data, $| T^m |$, and the number of decisions in the program

[5] All statistical methods and tables used are from [3].

under test, DE. We used linear regression to compute the parameters b_0 and b_1 in the following two equations:

$$| T^m | = b_1 * DE + b_0 \quad (3)$$

$$| T^d | = b_1 * DE + b_0 \quad (4)$$

Equation (4) was used for comparison with Weyuker's results. Table 5 lists the values of the parameters b_0 and b_1 for the two sets of experiments. This table also lists the weighted average of the number of test cases required, computed as in [18] using the formula:

$$\text{weighted average} = b_2 * DE \quad (5)$$

$$\text{where } \frac{1}{b_2} = \frac{1}{n}\sum_{i=1}^{n}\frac{DE_i}{T_i}, \; n \text{ is the number of programs}$$

The maximum values of $\frac{|T|}{DE}$ are also listed in Table 5.

Data in Table 5 lists the parameters obtained from a linear regression analysis between T^m and DE and T^d and DE. From Table 5 we see that multiplier b_1 is 1.72 and 0.6, respectively, in the linear equations expressing T^m and T^d for SET-I (see row 1 and row 2 of the table). This implies that a mutation adequate test set is about 2.9 times that of a data flow adequate test set. For SET-II this ratio is 4.5. These two ratios, 2.9 and 4.5, imply that for a given program, a mutation adequate test set is larger than a data flow adequate test set. The same conclusion is also reached by an examination of the values of the coefficient b_2 which is used in the weighted average computation. Also note that the maximum values of the ratio of the size of the adequate test set and DE (listed in columns 4 and 7 of Table 5) for both data flow and mutation is 4.0 and 4.5 for SET-I and 2.5 and 4.25 for SET-II.

From the data in Table 5 we conclude that the cost of both data flow and mutation is linear in the number of decisions in a program. However, the cost of mutation is approximately 2 to 3 times that of data flow. This is also confirmed from Table 3 where the average number of test cases required to satisfy the mutation criteria is 1.88 times that required to satisfy the data flow criteria.

A comparison of the regression data listed in Table 5 with that obtained by Weyuker is in order. In this discussion we refer to Table 1, column 5, page 124 in [18]. The least squares coefficient b_1 in Weyuker's experiments is 0.93 as compared to our estimates of 0.6 for SET-I and 0.32 for SET-II. The value of b_0 in Weyuker's experiments is 1.40 as compared to our estimates of 2.6 for SET-I and 3.77 for SET-II. Our results indicate a slightly lower cost of data flow than the cost estimated by Weyuker due to a lower b_1 and only a slightly higher b_0. As an example, for a program with 15 decision statements, a tester is expected to construct about 12 to 15 test cases to achieve ALL-DU path coverage using Weyuker's estimate. Using SET-I and SET-II data, our estimate of this range is 11 to 13 and 8 to 13, respectively. If, for the same program, a mutation adequate test set is required, then the tester is expected to construct 22 to 26 test cases as per our estimate using SET-I and 28 to 32 test cases using SET-II.

6.3 Correlating DE with other data

For each program we also examined the correlation between the number of decisions and the size of mutation and data flow adequate test sets, and the difference R between F_{T^m} and U_{T^d}. For this purpose we computed the Pearson's product moment correlation coefficients [3] for each of the two sets of experiments. The coefficients appear in Table 6. The first column in this table gives the average number of decisions in each set. The subsequent columns list the correlation between

Table 5: Results of regression analysis on the costs of data flow and mutation.

Test set	SET-I			SET-II		
(T)	$b_1 * DE + b_0$	$b_2 * DE$	$\max(\frac{1}{DE})$	$b_1 * DE + b_0$	$b_2 * DE$	$\max(\frac{1}{DE})$
T^m	$1.72 * DE - 0.12$	$1.52 * DE$	4.5	$1.46 * DE + 5.73$	$2.11 * DE$	4.25
T^d	$0.6 * DE + 2.6$	$.88 * DE$	4.0	$0.32 * DE + 3.77$	$.83 * DE$	2.5

Table 6: Correlation between the number of decisions in a program and various other statistics.

Set	DE	$\mid T^m \mid$	$\mid T^d \mid$	$F_{T^m} - U_{T^d}(R)$
I	7.33	0.86	0.85	0.35
II	6.40	0.87	0.73	0.47

DE and the item that labels the column. For example, 0.86 is the correlation between DE and the size of the mutation adequate test set in SET-I.

As expected, all the correlations in Table 6 are positive. From this data we observe that the correlation between DE and the adequate test sets, is high being 0.86 and 0.85 for SET-I and 0.87 and 0.73 for SET-II. This lends strength to our claim made earlier that the number of test cases required for obtaining mutation and data flow adequate data is linear in DE. Notice, however, that the correlation of DE with $\mid T^d \mid$ is smaller than its correlation with $\mid T^m \mid$. This indicates a stronger linear relationship between DE and $\mid T^m \mid$ than between DE and $\mid T^d \mid$. However, the difference is too small (0.87 versus 0.73 in SET-II) to make any strong conclusion. Further, no such correlation was reported by Weyuker against which we could compare our results.

An examination of the last column in Table 6 reveals a very low correlation between DE and R. This implies that program complexity, measured in terms of DE, does not significantly affect the relative difference between the strengths of the mutation and data flow criteria. This also indicates that the conclusions made above regarding the null hypothesis may be valid for programs with the larger values of DE usually associated with large (or more complex) programs.

6.4 Summary of statistical analysis

Below is a summary of our statistical analysis and conclusions based on the two independent sets of experiments. Our analysis and conclusions relate to the strengths of the data flow and mutation based test adequacy criteria and their relative costs.

1. Of the 18 programs used in our experimentation, for 11 (61%) programs mutation adequate test set also proved to be data flow adequate. On the contrary, the data flow adequate test set was also mutation adequate for only 1 (5.5%) program.

2. Both sets of experiments can be used to reject the null hypothesis as stated in (1) at 95% significance level using the sign-test and the Wilcoxon signed-rank test. For SET-II the hypothesis can be rejected at 99% significance level. This result lends support to the conjecture that for a given program, a mutation adequate test set is stronger than a data flow adequate test set.

3. A high correlation was found between the number of conditions in a program and the size of the mutation and data flow adequate test data. Weyuker has already provided data in

support of such a conjecture for data flow adequate test data. Our data lends support to the conjecture that even though the number of mutants generated [4] is in the order of n^2, n being the number of distinct program variables, the cost of mutation is linear in the number of decisions in the program.

4. Linear regression applied to pairs of values (DE, T^m) for both sets revealed that the coefficient of DE is less than 2. For data flow adequate test data this coefficient is less than 1. This lends support to the conjecture that the cost of data flow is lower than that of mutation. However, as indicated above, both costs are linear in the number of decisions in the program.

7 Avoiding possible pitfalls

As in all experiments conducted by humans, we noted the possibility of bias entering the final results. Such a bias could adversely affect the outcome of our experiments. We therefore took the following precautions to avoid such a bias:

1. *Data sharing*: At the start of the experiment it was decided that the two groups would not share any data other than the source programs. Data submitted by each group was reviewed periodically using parameters mentioned below, to ensure that this decision was adhered to. Within a group test data was shared as pointed out in Section 4. The generation of T^m and T^d was kept independent within each group.

2. *Multiple groups*: Instead of one group conducting the experiments, we decided that two groups conduct the experiment using identical methodology. This enabled us to compare the results obtained by one group with those by the other. The results from both groups exhibited a similar pattern. For example, as can be seen from the data presented in Section 4, both the groups found mutation adequate data sets to be consistently larger than data flow adequate whereas the opposite was not true except for two out of 18 programs.

3. *Program selection*: An overlapping, though not identical, set of programs was provided to both the groups. Thus, for example, for program P given to both the groups we compared the experimental outcomes. The comparison was based on values of two parameters: the size of the adequate test sets and the cross scores. Apparently random differences in the values of these parameters indicated that the two groups did not collaborate. For example, $\mid T^m \mid$ for the *triangle* program was 53 for SET-I and 39 for set 2. The corresponding cross scores were 0.8 and 0.81, respectively.

4. *Multi-test set sampling*: In our experiments, for each program, we constructed only one adequate test set for each testing method. However, in general, there exist several mutation or data flow adequate test sets for each program. Thus, instead of obtaining just one cross score per testing method for each program, one could also obtain a larger sample of cross scores and take its average as the representative cross score for that program.

   Generating multiple adequate test sets for each program, such that each one is independent of the other, is a resource intensive task. In the absence of any automatic test data generator, the only way such independence could be achieved is by employing several persons. Each person could then be assigned to generate one test adequate test set for data flow and one for mutation. It was the non-availability of the desired number of persons that led us to generate only one adequate test case per program per testing method. This could certainly

be construed as a weakness of our experiments. We however believe that the data that we have provided provides a basis for further detailed experimentation.

Perhaps the best test of the accuracy of the above remarks would be a repetition of the above experiment by a different research group. The tools and programs that we used in our experiments are all in public domain. Hence, if need be, such a repetition is indeed possible without any undue effort that would otherwise go into the making of these tools and for program preparation.

8 Concluding Remarks and Future Work

It is important to note that the observations and analysis presented above are derived from two samples consisting of 14 and 9 programs respectively. Further, the maximum number of decisions in any of these programs was 28. Thus these samples contain relatively small programs as compared to programs that are generally encountered in practice such as an editor, a compiler, an operating system, or a telephone switching system. Other than the reasons mentioned in the previous section, we do not have any reason to argue in favor of or against the view that statistical analysis would reveal significantly different results for larger programs. More experimentation and/or an in-depth theoretical comparison of the two methods is required to obtain a decisive answer to the conjectures mentioned above.

Our analysis indicates that a mutation adequate test set is stronger than a data flow adequate test set. Such a result, however, does not indicate how good the fault detection capability of mutation is as compared to that of data flow. In this regard, Budd's study appears to be the most authoritative up until now. In his dissertation [4], Budd compared the effectiveness of complete path testing against mutation testing. Recall that the criteria that constitute data flow testing are subsumed by the ALL-paths criteria [6]. Of the 22 faults analyzed by Howden it was shown [11] that path testing and symbolic execution combined would detect 13 of these faults. Budd showed that mutation will reveal 20 of these 22 faults. Budd compared the effectiveness of mutation using other data as well and found that path testing invariably could never find more errors than mutation. Thus Budd's results are indicative of the fact that, indeed, the fault detection capability of mutation is superior to that of path testing (and to that of data flow testing). Combining Budd's results, obtained deterministically, and ours, obtained statistically, we obtain strong evidence in support of the conjecture that mutation testing is superior to data flow testing both in terms of its adequacy criteria and the fault detection capability.

Once again, we would like to point out that both Budd's and our results are based on a small number of relatively short programs. However, our results strongly indicate that for unit testing mutation will be more effective than data flow in ensuring reliable units.

We are currently studying the fault detection capability of mutation and data flow using significantly larger programs that consist of over 100 conditions.

Acknowledgement

Elaine Weyuker provided us with ASSET. Rajiv Chaudhary, Duu-Iong Fang, Wei Tsu-min Huang, Stephen F. Maher, Maryann Perez, Lih-chyun Shu, and Wei Jen Yeh conducted the experiments reported here. R. J. Martin reviewed a preliminary version of this paper. Vernon Rego's expertise in statistical methods proved useful in our data analysis. Comments from the anonymous referees proved useful in improving the quality of this paper. My sincere thanks to all these people.

References

[1] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, Wm. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford, "Design of Mutant Operators for the C Programming Language," *Technical Report*, SERC-TR-41-P, Software Engineering. Research Center, Purdue University, W. Lafayette, IN 47907, 1989.

[2] V. R. Basili and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Trans. on Software Eng.*, Vol. SE-13, No. 12, December 1987, pp 1278-1296.

[3] G. K. Bhattacharya and R. A. Johnson, "Statistical Concepts and Methods," John Wiley & Sons Inc., New York, 1977.

[4] T. A. Budd, "Mutation Analysis of Program Test Data," Dissertation, Yale University, May, 1980.

[5] B. J. Choi, R. A. DeMillo, E. W. Krauser, A. P. Mathur, R. J. Martin, A. J. Offutt, H. Pan, and E. H. Spafford, "The Mothra Toolset," *Proceedings of Hawaii International Conference on System Sciences*, HI, January 3-6, 1989.

[6] L. A. Clarke, A. Podgruski, D. J. Richardson, and S. Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria," *IEEE Trans. Software Eng.*, Vol. 15, No 11, pp 1318-1332, November 1989.

[7] R. A. DeMillo, D.E. Hocking and M.J.Merrit, "A Comparison of Some Reliable Test Data Generation Procedures," *Technical Report*, GIT-ICS-81/08, Georgia Institute of Technology, Atlanta, GA 30332, 1981.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, Vol. 11, No. 4, April 1978.

[9] R. Hamlet, "Theoretical Comparison of Testing Methods," *Proc. Third Symposium on Software Testing, Analysis, and Verification*, Key West, FL, pp 28-37, December, 1989.

[10] J. R. Horgan and S. A. London, "ATAC- Automatic Test Analysis for C Programs," *Memorandum*, Bell Communications Research (Bellcore) internal memorandum, TM-TSV-017980, 1990. Morristown, NJ.

[11] W. E. Howden, "Reliability of the Path Testing Strategy," *IEEE Trans. Software Eng.*, Vol. SE-2, No. 3, pp 208-215, September 1976.

[12] B. W. Kernighan and P. J. Plauger, "Software Tools in Pascal," Addison-Wesley, 1981.

[13] A. J. Offutt, "Automatic Test Data Generation," Technical Report, SERC-TR-25-P, Software Engineering. Research Center, Purdue University, W. Lafayette, IN 47907, 1986.

[14] S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. Software Eng.*, Vol. SE-11, No. 4, pp 367-375, April 1985.

[15] P. G. Frankl, S. N. Weiss, and E. J. Weyuker, "ASSET: A System to select and evaluate tests," *Proc. IEEE Conf. Software Tools*, New York, April 1985.

[16] T. J. Ostrand and E. J. Weyuker, "Collecting and Categorizing Software Error Data in an Industrial Environment," *The Journal of Systems and Software*, Vol. 4, pp 289-300, 1984.

[17] "User's Manual Stat/Library$^{T.M.}$," IMSL Problem Solving Software Systems, IMSL, TX, 1987.

[18] E. J. Weyuker, "The Cost of Data Flow Testing: An Empirical Study," *IEEE Trans. on Software Eng.*, Vol. 16, No. 2, February 1990, pp 121-127.

APPENDIX

Table 7: Summary of terminology used

Symbol	Meaning
$\mathcal{D}$	Set of subsets of test cases in the input domain of P.
DE	Number of decisions in P.
F_{T^m}	Data flow score of a mutation adequate test set.
M	A mutant of P.
$\mid M \mid$	Number of mutants of P.
MS	Mutation score.
$M(t)$	Output obtained by executing M on test case t.
U_{T^d}	Data flow score of a mutation adequate test set.
p_x	A DU-path of variable x.
P	Program under test.
$P(t)$	Output obtained by executing P on test case t.
S_d^x	Statement containing a definition of variable x.
S_u^x	Statement containing a use of variable x.
T	Set of test cases used for testing P.
t	An element of T.
$\mid T \mid$	Size of test set T.
T^d	Data flow adequate test set for P.
$\mid T^d \mid$	Size of data flow adequate test set for P.
T^m	Mutation adequate test set.
$\mid T^m \mid$	Size of mutation adequate test set.

TESTING A GRAPHICAL USER INTERFACE,
Experiences with Automation

This paper describes our experiences in automating a test suite for a graphical user interface. We describe the process, tools, problems and benefits of automation.

Nancy K. Winston
Tamara Baughman

Mentor Graphics Corporation

8005 S.W. Boeckman Road
Wilsonville, Oregon 97070-7777
(503) 685-7000
Email: nwinston@MENTORG.COM
and
tamarab@MENTORG.COM

Biographical Sketches

Nancy K. Winston currently works for Mentor Graphics Corporation in the QA group on the Common User Interface project and as a project leader for several related projects. She is a recent recipient of the Mentor Graphics Chairman's Achievement Award. Prior to working at Mentor Graphics, Nancy developed real-time software and database systems.

Tamara Baughman has a B.S. in Computer Systems Engineering from the Oregon Institute of Technology. She has worked at Mentor Graphics Corporation for 5 years and is currently in the QA group on the Common User Interface project. She is currently investigating GUI industry standards for Mentor Graphics products.

Target Audience: Technical

Keywords: Graphical User Interface, Testing, Verification, Automation

Introduction

The verification of Graphical User Interface (GUI*)† software is always a challenge. In projects where it is desirable to set up an automated regression test suite for verifying the software multiple times during the project life cycle, the challenge is even greater.

A GUI development project typically includes iterative builds of the software system. After each build or release, user feedback is sought on the system appearance and behavior. It is frequently desirable to make changes quite late in the product development cycle in response to user feedback.

The process of verifying visual changes is problematic. The human eye is frequently inadequate for viewing small graphic objects on the screen. Some undesirable regressions to the visual images may mistakenly slip through an iterative release cycle if the development team relies solely on visual inspection. Also, visual verification of software releases is very costly.

A common method of verifying graphical software is to save images of correct screen displays, and compare screen displays from future builds to the saved images. Any changes to the appearance of the screens after capturing the reference screen images causes a miscompare. Verification scenarios which include this screen image compare technique make it difficult to accommodate desired late changes to the appearance of the screens. This is because all of the references must be updated to include the changes.

We recently went through the process of designing and releasing a new GUI at Mentor Graphics Corporation. We made the decision to automate as much of the verification process as possible. We developed a process for automation and the tools to support it. This process supports the functional testing for the GUI and does not address usability testing. Our toolset includes screen image comparators and graphical viewers to highlight the areas where two images compare and miscompare.

This paper describes our experiences and the lessons we learned from them. In it, we describe the process and identify the tools needed for the process. It also includes a discussion of why we made the decision to automate the process and the benefits we gained from it, identification of problem areas, our solutions to the problems raised, and areas for future improvement. Test case selection

† Starred terms are defined in the Glossary.

paradigms are not covered by this paper.

Development Model for the CUI Project

Mentor Graphics Corporation (MGC*) released a new generation of software this year. The release, Version 8.0 (V8.0), consists of Mentor Graphics applications rewritten using object oriented design techniques in C++, an object oriented language. Included in this V8.0 release is the Falcon Framework™*. This is a new piece of software that is incorporated into Mentor Graphics 8.0 applications.

The Common User Interface® (CUI*) layer of software is included in the Falcon Framework™. The CUI provides a Motif† compliant GUI for applications. This includes facilities for customizing menus, creating dialog boxes* with graphical controls*/widgets*, changing key definitions, and other application environment features.

As mentioned previously, MGC applications incorporate the Falcon Framework™ which includes the CUI. During the V8.0 development cycle, we employed parallel project development as much as possible. This meant that while the CUI was being developed, the applications relying on it were also being developed. This parallel model meant that the CUI team had internal customers within MGC for releases of software as well as having an end user customer base for the product.

Having internal customers depending on incremental CUI software releases greatly influenced the release cycle for the project. Frequently, each application had needs for releases and functionality that were not consistent with other applications. This led our group to make frequent internal releases to keep our application groups productive with the functionality they needed.

Timeliness of releases and functionality were not the only productivity factors for application development teams. They also needed high quality software so they could spend a minimal amount of time finding and working around CUI bugs. This requirement meant that each time the team made an internal software release available, it had to be verified in some manner.

† Motif is a Trademark of the Open Software Foundation

There were two more major factors that influenced the CUI project development cycle. First, usage paradigms of the newly designed products were not entirely clear at the project start. Since one of the main goals of a GUI is to make products more usable, doing usage studies of the product and iteratively refining it during development was a necessity. Second, the Motif standard was evolving as we developed the CUI. Iterative changes to meet the standard were required as the standard changed.

The frequent releases, incremental functionality deliverables, desire for high quality, unclear usage paradigms, and evolving standards led to a very dynamic development model for the CUI project. This model heavily influenced some of the verification design decisions and test suite usage.

Non-Automated Verification Procedures

When the CUI project started, the team performed test case development and verification on an ad-hoc basis. Shortly before the authors joined the project, the test case development was becoming more methodical and organized but the verification was still often fairly ad-hoc. It was also time-consuming. Verifying a release took between 1-2 person weeks depending on the number of problems found. By applying 2-3 people to the process, we completed release testing in 2-5 calendar days. In our development environment, that timeframe was unacceptable. The CUI project was frequently on the critical path for the Falcon Framework releases due to the verification timetable.

Regardless of whether or not the process is automated, the release verification task is divided into three parts. Existing test cases are executed and analyzed, test cases for new functionality are written and executed, and bug fixes are verified.

The CUI project has an example application that exercises the basic product functionality. Figure 1 illustrates the visual attributes of this example application.

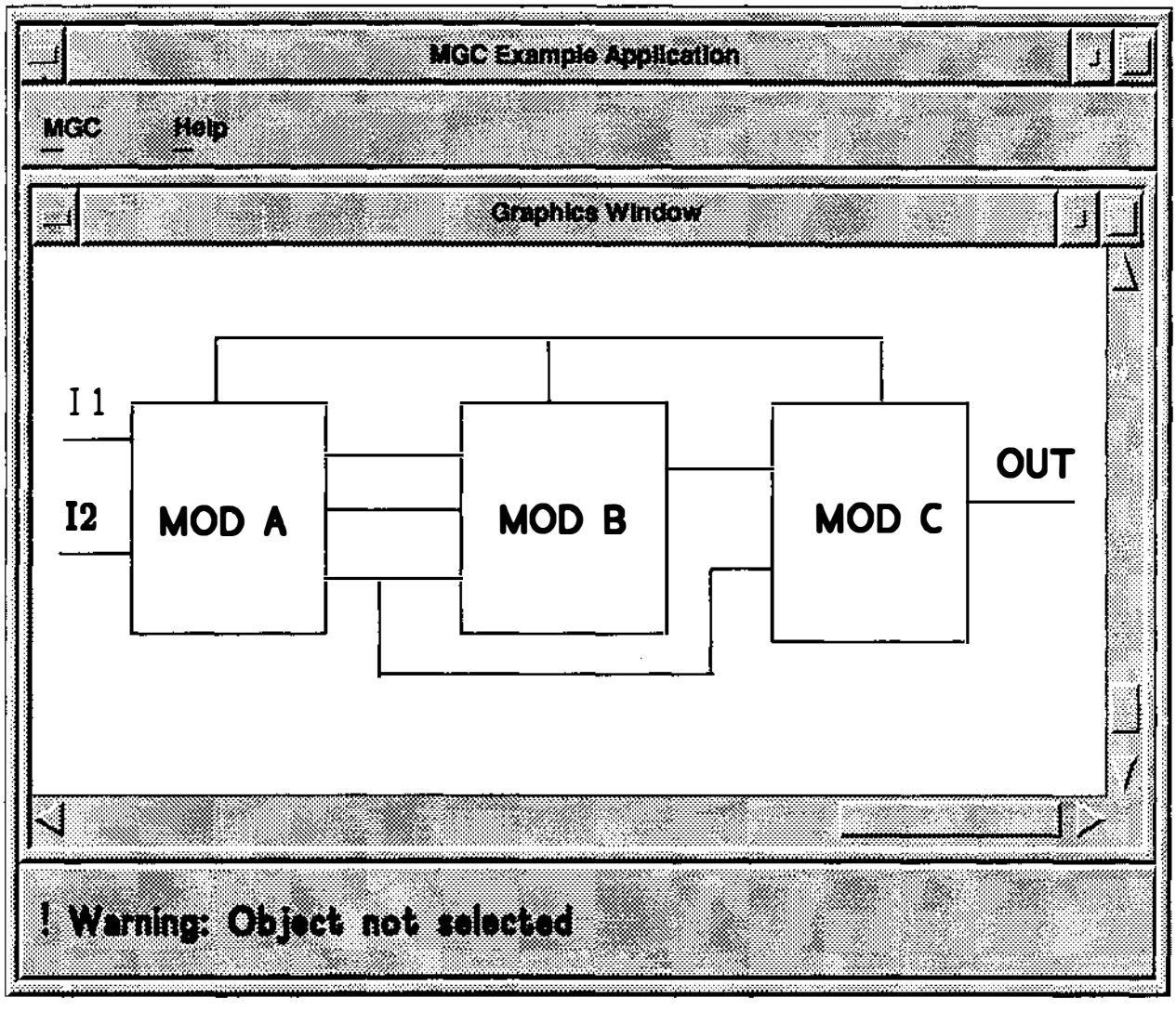


Figure 1 Example GUI Application

Prior to automation, an engineer would run the existing test suite by invoking the example program (see Figure 1), loading the test case into the example program, executing the test case, and verifying that the test ran correctly. Because the entire process, including verification, was manual, the engineer running the tests had to know quite a bit about each test in order to run it. It was necessary to know the purpose of the test, how to execute it, and what constituted a successful run versus a failure. This process made it most efficient for the test authors to also be responsible for test execution to minimize effort. That was not always practical or desirable. Other engineers were required to learn the test cases in detail, duplicating effort.

New test cases were usually required to verify new functionality in the releases. The schedules did not always allow time to formally write new test cases prior to shipping an internal release. New functionality in the CUI was often tested by "playing" interactively with the example application. For instance, if a new menu was added to the CUI, the engineer responsible for verifying it would invoke the example application, interactively cause the menu to display, and check to see that the menu looked correct and operated correctly. When the engineer closed the example application, that test case was lost. There was no automated way to re-execute the test on the next software version.

The third segment of release verification involves testing bug fixes. Prior to automation, test cases for bug fixes were written but had to be run manually in the same manner as described above. Frequently, there was not time to write these tests before releases, so the tests were performed interactively.

Without automation of the execution and verification of the tests, there was always a risk that test runs and results were inconsistent between releases. If something "slipped through" the process due to human error, it was possible to deliver problems to internal customers that translated directly into productivity losses.

Automation Description

o Automation Strategy Overview

In January 1990, we made a commitment to provide an automated acceptance test suite to a system software vendor. The vendor wanted to run our tests before delivering new operating system versions to MGC. We had long recognized the need for an automated test suite for internal releases. Scheduling its development was problematic due to frequent releases and a changing interface (which would obsolete bitmap* files). We started working on the automation development in late 1989 in order to meet our schedule to our vendor. We completed the acceptance test suite in January and had the regression environment set up in February.

The test environment consists of:

Test Drivers: There are shell script drivers for the acceptance test suite and the regression test suite. They invoke the CUI example program once for each test case that is run. The function invocation for test cases is accomplished using

redirected input into the example program. The window for the example program is created with the same screen coordinates for each test run.

Test Support Routines: There are additional files loaded into the example application. These are a series of Test Support Routines (TSRs*)† which are functions that aid in reporting the test run data. They also support function calls to the CUI with reduced argument lists and calls into some CUI testability hooks.

Bitmap programs: The test driver also performs screen image (bitmap) compares. The bitmap reference files stored in the acceptance and regression reference directories are compared to bitmaps generated during the test run. These bitmaps are generated using the CUI-provided screen image save functionality. The program bmap_comp is used to compare bitmaps and view bitmaps that miscompare. The program view_bmap may be used to view bitmaps.

File control for test runs: For each test run, the tester can specify an individual test case or several test cases. If several test cases are chosen for a test run, a file that controls which test cases to run is input to the driver. The file is formatted into lines that have the directory of the test case followed by white space followed by the test case name. The directory name correlates to a functional subsystem of the CUI. If just the directory is listed, all test cases ending in '.test' in that directory will be run.

Test Suite Conventions: There are a set of conventions that the test suite control programs depend on including names of files (prefixes and suffixes), directory structures, standard Unix® AT&T utilities' functionality and their location, having a standard color map installed and read/write permissions for the test directories.

Test cases: Test cases are specified in the MGC extension language, AMPLE™*. This language is part of the Falcon Framework™ and is "C-like". These test cases call the TSR functions for common routines such as error reporting. The test cases contain the actual test code for exercising menus, dialog boxes, etc. Each test case name ends in a ".test" suffix and contains a main test function named "ftest()". The test cases are loaded into the example program and executed via redirected input.

† See Appendix A for TSR examples.

Results Reporting: Each test case generates logical output text (logical transcript file) which details the functions called and their results. This output is saved into a file which is compared to a reference file by the test driver. Screen images are also saved and compared to reference image files (bitmap files). For a test run, there is a log file saved that details all test cases run and their high level results.†

o Details of Automation Strategy

Several tools were available to help us automate the test suite. We had some testability hooks available in the CUI which would allow us to record and playback the physical events. Examples of physical events are mouse clicks, cursor movement or key presses. We also had functionality in the CUI to capture screen images in a HP/Apollo GPR bitmap format. There had also been some programs developed inhouse to allow two HP/Apollo GPR bitmap files to be compared, viewed, and compressed. Because we already had the toolset for the HP/Apollo GPR bitmap format, we used this format as our starting point in the test suite.

The bitmap compare program provides the tester with the capability to view the differences between the files visually or in an automated fashion. Invoked visually, it clears the screen and alternately displays both bitmaps on the screen. Using a key sequence, the user can stop the alternate display and toggle between the two bitmaps or choose to just see the differences between them (an XOR comparison).

A Unix® AT&T shell script (test driver) was developed which controls the test run and the reporting of the test results. The shell script also invokes the CUI example program once for each test case that is run. Each test case is loaded into the example program using the facility provided by the CUI for loading extension language programs. The test driver also controls the window size and location for the test run. The window is created with the same coordinates for each test so the screen images are always the same size and in the same location for comparisons, and the physical events playback correctly. (The recording of the physical events is dependent upon the screen location of the mouse cursor.)

Due to the number of releases the CUI team had to provide, it was very important that the tests be set up so the tester could easily specify which tests to

† See Appendix C for a Test Run Log File example.

run. The tester could run the entire regression test suite, a particular subset of tests or just one test. This flexibility allowed the tester to run the exact number of test cases that were appropriate for the release. Each release can be analyzed† to determine which set of tests need to be re-executed to verify the release. This particular scheme also provides a framework for new test development. An additional benefit to this strategy was that it made running tests so easy, the development engineers started running tests to validate their coding changes before the QA engineers received the release.

Another requirement for the test suite was to provide a mechanism in which the tester could run the tests interactively (manual mode) or automatically. This is useful because there are times when it is necessary to execute a group of tests manually. For example, when using the CUI facility for capturing interactive sessions for future replay, manual execution allows the tester to simulate user actions, place them into a file, and then use the automatic mode for replaying them in order to verify the test case. Developers use the manual mode to debug reported problems. When verifying a release, it is faster to execute test cases in an automatic fashion and analyze the results after all tests have run.

Finally, in order to easily analyze the results of a regression test run, it was necessary to write a shell script which logged tests which did not execute correctly. The majority of the CUI test cases relied on the result of bitmap image comparisons for validation. Also, the CUI was in a very dynamic state. A bitmap miscompare could be the result of a deliberate change in the software or an actual bug. The results analysis program allowed the miscompared files to be visually inspected and the user was given the option to update the reference files. This reduced the amount of time needed to update reference files when deliberate visual changes occurred in the software. Figure 2 shows the flow of a typical automated regression run.

† The analysis uses data gleaned from code reviews and walkthroughs, team discussions, source code control system reports on file changes, and information from the project manager. These factors are all considered and the QA team decides which tests should be run.

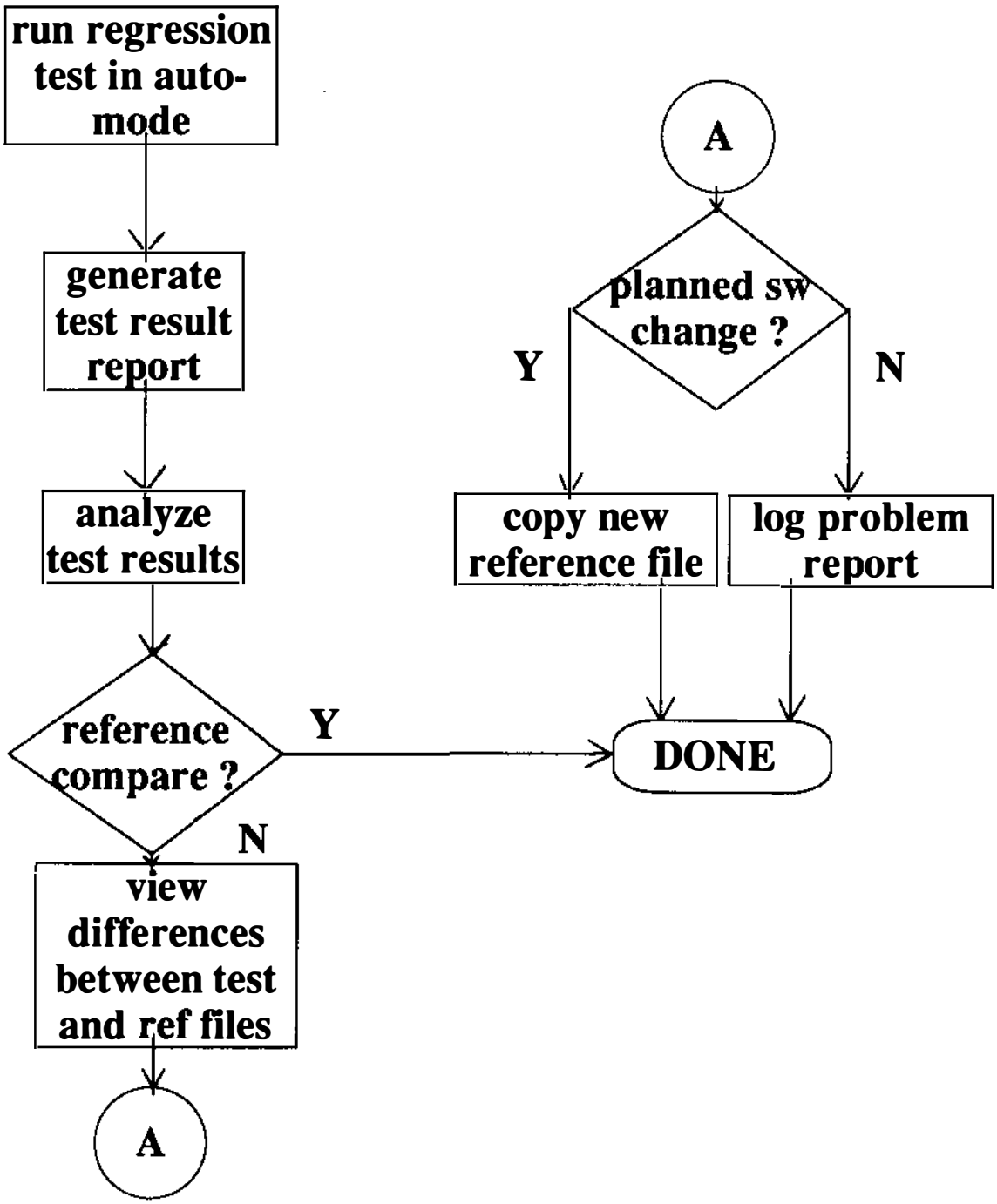


Figure 2

Based on the above assumptions and requirements, test case subdirectories were created which represent each of the functional subsystems of the CUI. For

example, there is a separate directory for the menu subsystem, the dialog box subsystem, etc. Each test case was placed in the directory of the subsystem it tested. This allowed for all the tests for a particular subsystem to be executed without running the entire test suite.

After it was determined how the tests would be organized, it was necessary to decide how to document and format each individual test case. The documentation consists of a title, a detailed description of the purpose of the test (what functionality is being tested), the test inputs and the expected output. Each test was written in the AMPLE™ extension language. Also, it may be necessary to input physical events which simulate interactive user actions to test the CUI (ie: press the mouse button on a menu item, drag the mouse, release the mouse button). The CUI has a record and playback facility called "physical transcripting*" that allows this type of input in an automated fashion. Many of the tests contain physical transcripts as part of the test input.

When a test case is written, the test needs to be debugged to ensure that it runs correctly. Each test contains a function (called manual_ftest()) which when executed, provides the tester with instructions for the test scenario and expected results. If the test requires a physical transcript, this function provides the instructions for capturing the physical events in a log for future replay†. Figure 3 shows the typical flow of test case development.

† See Appendix B for a Physical Transcript file example.

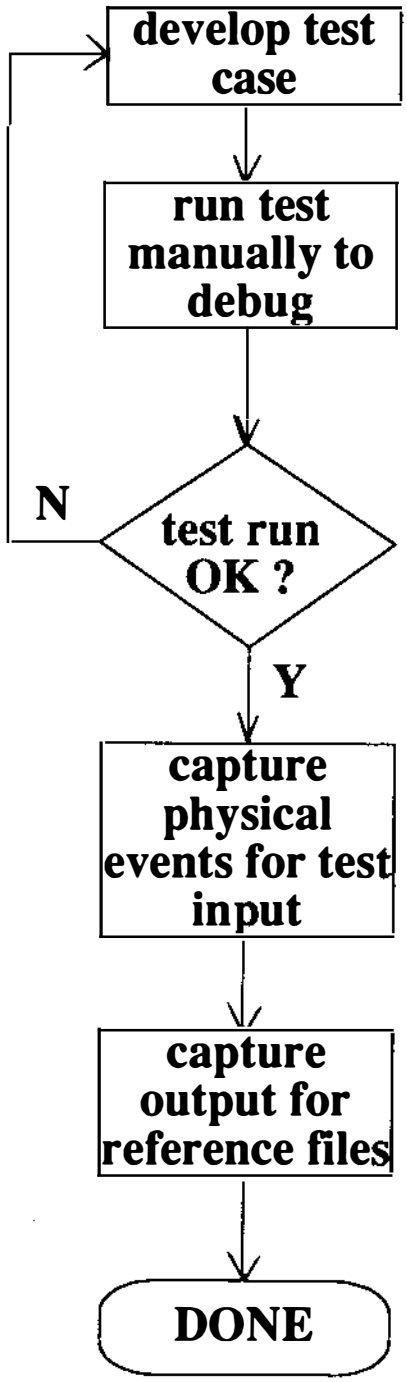

develop test case
run test manually to debug
test run OK ?
N
Y
capture physical events for test input
capture output for reference files
DONE

Figure 3

When verifying a release or debugging a problem, it is often desirable to have two different modes of test execution. manual_ftest() is the function which gets executed when the tester specifies a manual test run. ftest() is the function which runs the test case in an automated mode. If the input includes a physical transcript, it is replayed by this function.

The test output has two forms - physical screen images and a logical record of the test execution. The CUI sends logical data about the function execution to an output stream so that users of MGC applications have a history of their actions. The test driver redirects this logical transcript output to a test case log file which is compared to a reference test case log file†.

In some cases, it is possible to examine the internal state of the CUI. The software provides AMPLE™ system functions which return this type of information. When this is available, the test case software can check for expected values without having to capture a bitmap image. If the expected result was not obtained, then the test case can generate an error message using one of the TSR functions. The error message consists of an error number, the test result, the expected result and a description of the error. This allows the tester to later go back and determine exactly where the test failed. These error messages are captured in the logical transcript of the test run. When comparing the logical transcript to the reference file, the discrepancy will be brought to the tester's attention.

To test some graphical states, it is necessary to capture the graphical image of the screen. The CUI provides an AMPLE™ function which will generate a bitmap and save the data in a file. The amount of the image saved can be qualified by specifying the entire application window, a particular window within the application or a user defined area. This can help to reduce the size of the bitmap file. These bitmap files which are generated during a test run can then be compared with their corresponding reference files in order to verify the test run.

To support the development of the test cases, TSRs were developed. The TSRs were also written in AMPLE™. They are loaded into the example application along with the test cases. These TSRs provide a set of functions which represent repetitive tasks for each test case. For example: a function which would check that two arguments were equal and report an error message if they were not equal; a function which would take multiple strings and display them in the logical transcript for instructional purposes.

---

† See Appendix D for an example of these logical transcript log files.

There are still the same three release verification tasks to accomplish under the automation setup - running existing test cases, writing and running new test cases, and verifying bug fixes. The main difference is that the environment they are done under is different. The test runs can be controlled so that the same regression and bug fix tests are run on multiple releases giving a high degree of confidence in the releases. Also, the automation strategy allows us more time to write new functionality tests and bug fix tests prior to the release shipment.

Automation Savings on the CUI project

The initial cost of setting up the automated test suite was approximately two months for two QA engineers. However, the payoff was high considering the number of releases the project had to support. Verifying a release cost 1-2 person weeks prior to automation. Since the automated test suite was implemented, the cost averages 1-2 person days. That is an 80% reduction in cost. The savings are more than time, although that is always an important metric. Morale on the project has greatly increased because engineers are not always tied to manually running tests. There is much more time available for up-front quality activities like specification and code reviews and walkthroughs, requirements interviews, and technical training. We are able to find bugs faster resulting in less stress for the entire team. There is also much more time for test development and documentation. Also, our team is usually no longer the critical path for Falcon Framework™ releases.

Problems With Automation

The CUI automation strategy included saving screen images and comparing them to saved images for an exact match. The development cycle for the CUI was iterative, with the team making changes to the appearance and behavior based on usage feedback. When changes to the appearance were made deliberately, it became necessary to update all of the screen image reference files to conform to the new appearance. Also, some of the visual changes altered the real-estate of the CUI. The CUI record/playback functionality is dependent on physical locations relative to the screen. Real-estate changes caused some physical transcripts to replay incorrectly because the graphical objects that were being acted on had moved. This meant re-logging the physical events for those tests. This process was time consuming and tedious.

Our solution to this was to group visual changes into a few specific software releases. The downside to this was that we were less responsive to change than our customers would have liked. We received comments from them stating that they wanted us to wait until further in the development cycle to use this type of automation so that we could be more responsive.

The automation strategy did not account for verifying multiple Apollo display types. Multiple display type verification occurred infrequently and it was not cost effective to maintain several sets of bitmap reference files. So when performing configuration testing it was necessary to rely on the results of the logical transcripts and perform some of the more intensive visual tests manually. If this testing had been performed more frequently, we probably would have invested in automating it more fully.

Some testing hooks were specified to be added to the CUI software to aid automation. The priorities of this work conflicted with some other project goals. Some testing strategies were never implemented because these hooks were missing. These hooks should have been given equal priority to customer-visible functionality in the product specification. To address this problem, testing hooks are now included in design documents of all current and future CUI projects.

When the project started, MGC software was only delivered on one vendor's hardware. MGC now delivers software on multiple platforms. Changing platforms typically requires saving a new set of screen image reference files due to differences in rendering on different platforms. We are also changing the toolset to generate and read bitmaps with a TIFF format which is a more portable screen image format. There were some other portability issues with the test suite such as pathnames and filename length, which we modified the tests to accommodate. It took approximately three weeks to port the test suite to the SUN workstation. It is anticipated that it will take even less time to port to the next platform since all Apollo specific features have been removed from the test environment. The test environment is now a true Unix® environment with TIFF formated screen images.

Initially, the test cases were not placed under source code control. It was difficult to test different versions of software releases without version control on the tests. This work is currently underway.

Future Improvements Needed

After completing this project, it is clear to us that the overhead for this automation strategy would have been prohibitively high in the very early CUI project stages - from prototyping to early designs being available. We need other techniques to verify early graphical designs. We would like to investigate methods of object self-reporting. Each object would be required to report its current status. We would verify test objects visually and query them too. If the responses from the queries match the visual state observed, we might be able to extrapolate that additional verification of these objects could be accomplished through the query mechanism instead of capturing screen images. This would be a major step forward in our process.

We would also like to investigate some "fuzzy" comparison techniques. Perhaps we would be able to identify sections of the screen images that we don't want compared. It would also be nice to pick sections where certain ranges of values are close enough without matching exactly. We haven't done any work in this area yet.

After problem reports are filed due to test case failures, it would speed up verification of future test runs if these tests that are now "expected" to fail were flagged differently from new failures. We are currently looking into adding this expected failure condition to the test suite.

Physical transcripts are files that contain pairings of a physical screen location and an event. They are difficult to read and edit. If part of a transcript session needs to be modified, it is usually necessary to re-record the entire session. An interactive editor for these transcripts would be a helpful piece of functionality for the QA groups using this mechanism.

Glossary

- **Ample™** - Advanced Multi-Purpose Language. The Falcon Framework™ "C-like" extension language. With it, you can extend and customize the Common User Interface of MGC applications.

- **Bitmap** - A screen image.

- **CUI** - Common User Interface - Mentor Graphics Version 8.0 implementation of a Graphical User Interface. Alternately, a generic term equivalent to Graphical User Interface.

- **Dialog Box** - A rectangular transient object that provides information to the user or requests information from the user. Dialog boxes are often thought of as "forms" to fill out.

- **Falcon Framework™** - A common environment in which MGC applications run. MGC applications use the framework to provide a common interface, flexibility, and concurrency.

- **Graphical Controls** - Graphical objects within an application that allow users to interact with the application by directly manipulating them.

- **GUI** - Graphical User Interface - Provides a user with direct manipulation of graphical objects as a means of interacting with an application.

- **MGC** - Mentor Graphics Corporation - A company headquartered in Wilsonville, Oregon that produces computer solutions for CAE/CAD designers. Applications include software tools supporting schematic capture, integrated circuit layout, digital and analog simulation, printed circuit board layout, and electronic packaging.

- **Physical Transcript** - A file that contains pairings of a physical screen location and an event. This is the CUI physical event record and playback system data file.

- **TSRs** - Test Support Routines - Functions that aid in reporting the test run data.

- **Widget** - A graphical control.

Appendix A - Test Support Routine Examples

This appendix contains 4 example AMPLE™ code TSR functions and some AMPLE™ variable initializations. These are provided to illustrate the nature of AMPLE™ as well as some examples of the functionality that the TSRs provide.

```
extern area@@$apd_test_log_file      = $stderr;
extern area@@$apd_test_error_count = 0;
extern area@@$apd_test_case_count  = 0;

// ------------------------------------------------------------------------------
//   Print an error message to the log file and increment the error counter.
// ------------------------------------------------------------------------------
function $log_error_message( test_case_number : integer,
                             text             : rest
                           ), INVISIBLE
{
  area@@$apd_test_error_count = area@@$apd_test_error_count + 1;
  local hdr_msg = $strcat( $strcat( "Test case " ,
                                    $i(test_case_number, 5, , , @zero)  ),
                                      " : BUG detected" );
  $writeln_file($stdout, hdr_msg);
  if (area@@$apd_test_logging_on)
    $writeln_file(area@@$apd_test_log_file, hdr_msg);

  local num_lines = length(text);
  local i;
  local prefix = "";
  for (i = 1; i = num_lines; i = i + 1)
    {
      $writeln_file($stdout, $strcat( prefix ,text[((i) - 1)] ));
      if (area@@$apd_test_logging_on)
        $writeln_file(area@@$apd_test_log_file,
                      $strcat( prefix ,text[((i) - 1)] ));
      prefix = "   ";
    }
}

function increment_test_case_count()
{
  area@@$apd_test_case_count = area@@$apd_test_case_count + 1;
```

```
}

// ------------------------------------------------------------------------------
//   Check that the two input arguments are equal and print an log the error
//   if they aren't.
// ------------------------------------------------------------------------------
function $eq_check ( arg1,
                     arg2,
                     test_case_number : integer,
                     error_desc       : optional string { default = "" }
                   ), INVISIBLE
{
   increment_test_case_count();
   if ( arg1 != arg2 )
   {
      local err_msg = $strcat( $strcat( $strcat(
                                          "Test value (" ,arg2string(arg1) ),
                                ") != " ),
                       arg2string(arg2) );
      $log_error_message(test_case_number, err_msg, error_desc);
      return false;
   }
   return true;
}

// ------------------------------------------------------------------------------
//   Display helpful messages to the user in a separate window or dialog
//   box.
// ------------------------------------------------------------------------------
function $display_text( text : rest ),  INVISIBLE
{
   local num_lines = length(text);
   local i;
   // Send message to $stdout
   for (i = 1; i = num_lines; i = i + 1)
      $writeln_file($stdout, text[((i) - 1)]);
}
```

Appendix B - Physical Transcript File Example

This is an example of a physical transcript file, recorded interactively in the CUI. The test scenario for these physical events instructed the user to bring up a popup menu using the mouse and select a menu item.

```
( 141 , 311 ) :: 290126 Mouse_move
( 141 , 313 ) :: 290180 Mouse_move
( 142 , 307 ) :: 290196 Mouse_move
( 144 , 287 ) :: 290202 Mouse_move
( 146 , 259 ) :: 290207 Mouse_move
( 149 , 236 ) :: 290213 Mouse_move
( 150 , 226 ) :: 290218 Mouse_move
( 151 , 224 ) :: 290224 Mouse_move
( 152 , 221 ) :: 290230 Mouse_move
( 152 , 220 ) :: 290254 Middle_mouse_button [NoModifier] [down]
( 152 , 220 ) :: 290346 Middle_mouse_button [NoModifier] [up]
( 152 , 220 ) :: 290364 Right_mouse_button [NoModifier] [down]
( 152 , 220 ) :: 290374 Mouse_stop
( 159 , 223 ) :: 290750 Mouse_move
( 180 , 233 ) :: 290756 Mouse_move
( 194 , 241 ) :: 290761 Mouse_move
( 195 , 240 ) :: 290813 Mouse_move
( 195 , 240 ) :: 290814 Right_mouse_button [NoModifier] [up]
( 195 , 240 ) :: 290885 Mouse_stop
( 195 , 240 ) :: 291316 Other_key 9 [NoModifier] [down]
( 195 , 240 ) :: 291331 Other_key 9 [NoModifier] [up]
```

Appendix C - Logical Transcript from a Test Run

This transcript is output from a test run of 2 test cases from different subsystem directories - the forms and acceptance tests subsystems. It illustrates the format of results reporting. This is the highest level log file from the test run. The <filename>.tx files contain more detailed information about any failures reported and are the actual test case log files that are compared to reference test case log files. Note that the test, uw_session_area.test, had a bitmap comparison failure and an internal test case check failure. To investigate further, we would read the file uw_session_area.tx. After this test run, the user would also be prompted to run the bitmap comparison tool and the tools to update the reference files.

```
Mon May 13 08:58:15 PDT 1991
============================================================
---------------------------- CUI Regression Test ----------------------------
============================================================

User: nwinston, Node Name: tamarab
Note!  Total Number of Tests to Run = [      2 ]

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
(( 2 )) --> FORMS.HM/UW_NAMED_ARGUMENT.TEST >
                    /idea/tmp/uims_regr.uw_named_argument.tx

Started at 08:58:22
Stopped at 08:59:53

*** TEST EXECUTION SUMMARY:
> "Test session begins. ""Monday, May 13, 1991  08:59:06"
> "Begin test UW_NAMED_ARGUMENT"
> "End of test UW_NAMED_ARGUMENT"
> "  10 test cases"
> "   0 bugs detected."
> "Test session ends. ""Monday, May 13, 1991  08:59:42"

The complete test transcript is in /idea/tmp/uims_regr.uw_named_argument.tx

BITMAP compare ..... Passed
BITMAP compare ..... Passed
TRANSCRIPT compare ..... Passed
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
(( 1 ))  --> ACCEPT.HM/UW_SESSION_AREA.TEST >
                    /idea/tmp/uims_regr.uw_session_area.tx

Started at 09:15:02
Stopped at 09:16:05

*** TEST EXECUTION SUMMARY:
> "Test session begins. ""Monday, May 13, 1991  09:15:33"
> "Begin test UW_SESSION_AREA"
> "Test case 00003 : BUG detected"
> "Test value (false) != true"
> "   The sample window should be visible.  Expected failure PR24338"
> "End of test UW_SESSION_AREA"
> "   3 test cases"
> "   1 bug detected."
> "Test session ends. ""Monday, May 13, 1991  09:15:55"

The complete test transcript is in /idea/tmp/uims_regr.uw_session_area.tx

BITMAP compare ..... Passed
BITMAP compare ..... FAILED <<<<<<
BITMAP compare ..... Passed
TRANSCRIPT compare ..... FAILED <<<<<<

==========================================================
---------------------- CUI Regression Test COMPLETE ----------------------
==========================================================

Statistics Summary:
     13 internal test cases were exercised
    of which 1 (1%) failed.

    7 result files were generated
    7 of these result files were compared to reference files
   and 2 (29%) of these comparisons failed.

Regression Run STARTED :  08:58:15
      FINISHED:  09:16:15

==========================================================

Total Elapsed time -----> 00:18:00
```

APPENDIX D - Reference Test Case Log File

This is the test case log file that is the reference for the test uw_session_area.test in Appendix C. If the test in Appendix C had run correctly, it would have had the following output in the file named uw_session_area.tx.

```
ftest();
// "Begin test UW_SESSION_AREA"
$set_active_window("Transcript");
// "End of test UW_SESSION_AREA"
// "  3 test cases"
// "  0 bugs detected."
$set_active_window("sample");
```

User Interface Evaluation in the Real World: A Comparison of Four Techniques

Robin Jeffries
Hewlett-Packard Laboratories
P. O. Box 10490
Palo Alto, CA 94303-0969

Abstract

A user interface for a software product was evaluated prior to its release by four groups, each applying a different technique: heuristic evaluation, software guidelines, cognitive walkthroughs, and usability testing. The relative advantages of all the techniques are discussed and suggestions for improvements in the techniques are offered.

Author's biography

Robin Jeffries is a project manager in the Human-Computer Interaction Department of Hewlett-Packard Laboratories. She has a Ph.D. in Cognitive Psychology. She primarily conducts and directs research on the design of improved user interfaces for information management; her work on user interface evaluation grew out of an interest in improving the design of user interfaces by focusing on evaluation earlier in the design process.

Introduction

An important aspect of software quality is an application's usability, which can be loosely defined as how usable and useful the application, via its user interface, is to its intended consumers. Determining the usability of an application is surprisingly hard. There are to date no practical, analytic techniques that enable us to predict with any confidence the usability of a specific interface or to isolate the factors that contribute to a less than optimal user interface. All we have is a grab bag of heuristic methods that can expose some potential problems. The time honored method for determining the quality of the user interface is via usability testing, where a representative sample of prospective users are asked to do a variety of tasks using the application, and a trained observer makes inferences about problems in the user interface, based on difficulties the users have doing the tasks.

While usability testing has been quite successful in isolating problems with product user interfaces and feeding them back into the development process, there are some limitations to the technique. For one, it doesn't find all the problems in an interface. Products that have undergone extensive usability testing still generate complaints by users about the user interface. This is unavoidable; in an application of any complexity, it would be impossible to sample all of the tasks, possible actions, types of users, etc., that can impact the usability of the interface. Thus, only a subset of the possible problems will surface in any real test.

A second problem with usability testing is that the application must be quite complete before the evaluation can be done. This means that changes to the application of any significance are expensive to implement and can have a serious impact on a delivery schedule. Current wisdom has it that correcting a usability problem during the implementation or test phase costs a factor of ten more than correcting that same problem during the design or paper prototype phase. Of course, not all usability problems can be observed with mockups or prototypes of the user interface, but even if a subset of the problems could be corrected earlier, substantial savings would accrue.

The third problem with usability testing is that it is expensive to carry out. The costs include the special equipment (a testing lab) needed to conduct the studies, the rather substantial commitment in time to do the tests, and the special skills needed to design, conduct, and analyze data from such studies. Professionals with the appropriate skill and knowledge to do usability testing are a scarce resource in most organizations. Since they are typically assigned to several projects simultaneously, meshing their schedules with the needs of the product team is a problem.

Recently, several additional techniques have been proposed that would overcome some of the limitations of usability testing. The first of these is to have the same user interface professionals do more informal evaluations of the interface, by studying it in depth and looking for properties that they know, from experience, will lead to usability problems. We call this ***heuristic evaluation***, after [6]. Nielsen and Molich [6] showed that one could compensate for the idiosyncratic results one gets from a single heuristic evaluation by combining the results of several independent evaluations.

A second technique is the use of ***guidelines***. Guidelines provide evaluators with a set of criteria that the user interface must adhere to. Some types of guidelines are meant to check compliance to the standards and conventions of a particular interface style. These are commonly called style guides. Another class of guidelines is intended to measure adherence to generally accepted and broadly applicable principles of good user interface design (e.g., how the contents of a screen should be organized or how items should be arranged in a menu [1] [7]). It is this latter class of guidelines

that we refer to in this paper. Guidelines can be used by evaluators with no special training in user interface evaluation, if they include techniques for measuring how well a specific example adheres to each rule. Thus, the developers themselves could use appropriately constructed guidelines to critique their own designs. For the experiment described below, we used a set of 62 guidelines developed within Hewlett-Packard [2], that are written to be applied by software developers or evaluators without special user interface training.

A third technique is *cognitive walkthroughs*. The cognitive walkthrough method [5] combines software walkthroughs with a cognitive model of learning by exploration [4]. The evaluators walk through the interface in the context of core tasks a typical user will need to accomplish. The actions and feedback of the interface are compared to the users' goals and knowledge, and discrepancies between the user's expectations and the steps required by the interface are noted. The cognitive walkthrough method is intended to be used by the developers themselves and can be applied to early prototypes of the application. While this technique was recently developed in an academic environment and is not in common industrial use, the ideas embodied in it are interesting enough to consider as a alternative to more established techniques.

Little is known about how well any of these techniques work, especially in comparison to one another: what kinds of interface problems they are best-suited to detect, whether developers who are not user interface specialists can actually use them, and how they compare in cost/benefit terms. The experiment described in this paper was designed to provide such a test.

The Experiment

We obtained a pre-release version of a forthcoming software product and organized groups to evaluate its interface with the four techniques described above: heuristic evaluation, usability testing, guidelines, and cognitive walkthroughs. Each of the evaluation groups reported the problems they encountered on a common form, so that the numbers and kinds of problems detected by the different groups could be compared.

A primarily goal for this experiment was for all the evaluations to occur in conditions as close as possible to those that might exist in a real product evaluation. We used the results from the actual usability tests that were done for this product. Similarly, we used researchers in HP Laboratories who are frequently called upon to perform heuristic evaluations for real clients. The set of guidelines and the cognitive walkthrough technique have not been used in their current form on enough real products to determine what realistic usage patterns would be. Accordingly, we worked with the developers of these methodologies to set up procedures that were consistent with the ways that the technique developers intended them to be used.

The goal of realism means that these evaluations suffered from all the real-world problems of interface evaluation. We were limited in the number of people who could participate in the evaluations and the amount of time they could devote to them. We had limited access to the developers, who were hundreds of miles away and busy producing the final version of the product. However, we believe that these evaluations are quite typical of what goes on in product development, and therefore our results should be a good measure of how these techniques will work when applied in the real world.

Based on [6] we decided that four heuristic evaluators would provide a reasonable coverage of the set of problems in the interface (they recommend 3–5 heuristic evaluators to overcome the idiosyncratic focus of any individual evaluator).

Both the guidelines and the cognitive walkthroughs are intended to be used by the actual designers and implementers of the software being tested. Since we did not have access to the original designers for this study, we used teams of three software engineers. We chose them to be as similar to the actual developers as possible — they all had substantial familiarity with the platform on which the interface was built and had designed and implemented at least one graphical user interface. Since the biggest difference between them and the actual developers was familiarity with the application being evaluated, we asked all the evaluators to spend as much time as they could familiarizing themselves with the application before doing the evaluation. (For additional details on the experimental procedures, see [3].)

Results

The evaluators filled out a total of 268 problem report forms (152, 38, 38, and 40 for heuristic evaluation (HE), usability testing (UT), guidelines (G) and cognitive walkthroughs (CW) respectively). We sorted out several categories of reports that were not directly attributable to the interface being evaluated (e.g., problems caused by conventions or requirements of the underlying platform, or clear evaluator errors) and duplicates within groups (primarily multiple heuristic evaluators finding similar problems). This left a total of 206 core problems: HE: 105, UT: 31, G: 35 and CW: 35.

In terms of raw numbers of problems found, heuristic evaluation by four independent evaluators finds substantially more problems than do any of the other techniques. However, not all user interface problems are equally serious. Perhaps the heuristic evaluations were finding a large number of relatively trivial problems. To test this we had seven individuals rate the severity of the 206 core problems on a scale from 1 (trivial) to 9 (critical). The raters included four user interface specialists and three people with a moderate amount of such experience. The overall mean of 3.66 is indicative of an application whose user interface is relatively mature — the vast majority of truly serious problems had been found and fixed by previous evaluations in the development process.

	Heuristic Evaluation	Usability Testing	Guidelines	Cognitive Walkthroughs
Mean severity	3.59	4.15	3.61	3.44

Table 1: Mean problem severity by technique.

The mean ratings of the different groups [see Table 1] varied significantly. ($F(3,18) = 5.86, p < .01$). The higher rating for usability testing may reflect the ability of that technique to find more serious problems, or it may instead reflect a bias on the part of the raters. While evaluators in the other groups stated their problems in personal or neutral terms, the usability tester used phrases such as "users had trouble...". Thus, it was easy to tell which problems came from the usability test. Of course, attributing greater severity to problems reports that are backed by data is a reasonable strategy, both for our raters and for developers receiving usability reports.

A different notion of problem severity is the certainty that a problem really is one in need of repair. For instance, a missing label marking a numeric field as measuring minutes or seconds is clearly a problem needing to be fixed; the suggestion that animation would be a better way to show hierarchical directory relationships than a tree diagram is more a matter of taste. Preliminary results from ratings of this "real vs. taste" dimension suggest that the heuristic evaluation and cognitive walkthrough groups included more "taste" problems. If confirmed, this finding may reflect the inclusion criteria of different evaluators, or it may suggest something more fundamental about these evaluation techniques.

We also looked at the numbers of most severe (top third) and least severe (bottom third) problems found by the different groups. The heuristic evaluators found a much higher proportion of the least severe problems (almost twice as many least severe problems as most severe); however, in terms of raw numbers, the heuristic evaluators still found more highly severe problems than any of the other groups.

We used the severity scores and information about the time spent on each technique to produce a very rough benefit/cost analysis across the four techniques. Our model was that benefit could be approximated by the sum of problems found weighted by their severity, and cost by the time required to find those problems. This, of course, fails to take into account a number of aspects of the costs and benefits of doing evaluations. Some of these are measurable — e.g., the direct monetary costs of the evaluations, ranging from the salary of the evaluators to the cost of equipping a usability test lab, while others are more intangible — e.g., the benefit of having the evaluators do their own evaluations. However, we believe that even this simple model gets at some important aspects of costs and benefits of doing user interface evaluations.

Even within this model, we debated various ways of measuring the costs and benefits. For benefits, we considered both a linear weighting of problems by their rated severity and an exponential weighting based on rated severity, which gives much heavier weight to the most severe problems. For costs, we considered whether to include the time to be trained on the technique (for guidelines and heuristic evaluation); one could assign that cost completely to the first evaluation done with a technique or amortize it over a large number of future evaluations. We also debated whether to include interface familiarization time for the techniques that were intended to have been applied by the developers themselves. In a non-experimental situation, such costs would not exist; however, there were very different amounts of time spent by different groups, and we believe that those differences do impact the number of problems found.

We ended up doing a large number of different analyses, based on various combinations of assumptions about the appropriate factors to include in both costs and benefits. Table 2 gives the information needed to compute any of the benefit/cost ratios. The final row of the table gives the range of benefit/cost ratios that we found. While differing assumptions do change the results, a general trend is clear: under all assumptions, usability testing is the most expensive technique to apply, heuristic evaluation was the most cost effective, and guidelines and cognitive walkthroughs were consistently in between.

Comparing the Techniques

Overall, the heuristic evaluation technique produced the best results. It found the most problems, including more of the most serious ones, than did any other technique, and at the lowest cost.

	Heuristic Evaluation	Usability Testing	Guidelines	Cognitive Walkthroughs
	Benefit: Sum of severity scores			
linear	433	133	130	120
exponential	355	175	205	81
	Cost: Time spent on analysis (person-hours)			
analysis time	20	199	17	27
technique training	–	–	5	10
interface training	15	–	64	6
	Range of benefit/cost ratios: severity/time			
	12–10	1	12–2	4–2

Table 2: Benefit/cost ratios for the four techniques.

However, it is dependent upon having access several people with the knowledge and experience to effectively apply the technique. Our heuristic evaluators were skilled user interface professionals, with advanced degrees and years of experience in evaluating interfaces. They were also knowledgeable about the specific style of interface being evaluated and the task domain it applied to. Such people are a scarce resource, and their time is valuable, especially since multiple evaluators are necessary to obtain the kinds of results found here; no individual heuristic evaluator found more than 42 core problems. Other limitations of heuristic evaluation are the relative numbers of low priority problems reported and the tendency to report more problems of an idiosyncratic, "personal taste" nature. The kinds of problems found by these heuristic evaluators would primarily require a full functionality prototype. There are forms of heuristic evaluation that can be applied to early mockups of the interface; however, one should be cautious in generalizing our results to those very different forms of evaluation.

Usability testing did a good job of finding serious problems: on the average, the problems it found were more severe than the three other techniques, and it reported virtually none of the low priority problems that the heuristic evaluators focussed much of their energies on. However, it was the most expensive of the four techniques to apply. A formal usability test is by necessity quite time consuming. On the other hand, one must also consider the intangible benefits of the results: the necessity of fixing the problems found is practically indisputable; the experiences of real users back the conclusions drawn, rather than opinions, as is the case for heuristic evaluation. Thus, there is little additional cost directed at convincing developers to act on the findings. Formal usability tests occur late in the development process, requiring relatively complete and robust prototypes. But there are forms of user testing that can be conducted earlier in the development cycle with mockups. Again, the results of this experiment would not necessarily generalize to those techniques.

While guidelines overall produced a moderate benefit at a moderate cost, their most important role may be their ability to serve as a focusing device, forcing evaluators to take a broad look at the interface, rather than limiting their evaluation to a subset of the interface's properties. Such was the case here: in the post-evaluation questionnaire, the guidelines evaluators were more confident that they had covered the entire interface than the heuristic evaluators. Furthermore, the

guidelines evaluators indicated that many of the problems they found were not as a direct result of the guidelines themselves, but were unrelated problems they noticed in the process of applying a specific guideline. Guidelines have the advantage of being usable by the developers or other people not specially trained in user interface evaluation. They can only be applied, however, to the completed interface, late in the development process. The same guidelines could, in principle, be used by developers at the time they were designing an interface, to help them make more informed design choices. We have not examined whether developers could use the advice in the guidelines to produce better user interfaces, but it may be a promising, low-cost approach.

The cognitive walkthrough technique needs further refinement before it is generally useful for user interface evaluation. The concept is very attractive: a technique that can be used by the designers themselves on very early prototypes or designs to critique the interface in the context of the actual intended tasks and users. An especially important benefit would be that the developers' understanding of the source of the problems should eliminate the need for intermediaries to translate the problem descriptions into recommendations for redesign.

The results of this experiment showed that the walkthrough technique partially lives up to these expectations. Software designers similar to the developers of this interface were able to use the technique successfully and found a moderate number of problems. However, there were many limitations to their use of the technique. First, the successful use of the cognitive walkthrough technique critically depends on the selection of an appropriate set of tasks. Our pilot testing showed that evaluators were unable to devise suitably representative tasks for this evaluation; thus the experimenters constructed the actual tasks used in the experiment. Currently, selecting tasks for complex interfaces such as this one requires substantial expertise; the technique realistically could only be applied if the developers worked with a user interface specialist for that part of the evaluation. Second, the walkthrough evaluators were only able to complete seven tasks during their evaluation sessions, which consumed as much time as we could convince them to spend on the evaluation. More tasks would be necessary to cover all of this interface (which supports a particularly broad class of tasks). Especially since the evaluators all found the application of the technique to be tedious, it is not clear whether evaluators could be persuaded to put a sufficient amount of time into this type of an analysis. All of these problems are being addressed by the developers of cognitive walkthroughs, and future versions of the technique should minimize some of these difficulties.

One advantage of the walkthrough technique that was noted by all of the walkthrough evaluators was that the outputs of the walkthrough — enumerations of the knowledge that users are assumed to have and the internal states of the system that are relevant to users' interaction with it — would be of significant value to designers and other members of a development team, such as documentation writers. This may be the most useful aspect of the walkthrough technique, and needs to be critically evaluated.

Conclusions

Three different dimensions of an evaluation technique need to be considered when devising an evaluation strategy for a particular application: who will do the evaluation, at what stage in the development process will the evaluation be done, and what are the costs associated with a particular

evaluation technique. The four techniques considered in this experiment vary along all three of those dimensions.

Access to individuals with particular skills and with time available to do the evaluation is an important prerequisite for certain techniques. Heuristic evaluation and usability testing require highly trained specialists; this is especially true for heuristic evaluation, where multiple independent evaluations are required. Small organizations may not be able to afford the overhead of having such specialists available. Cognitive walkthroughs and guidelines can be applied by non-specialists who are familiar with the interface. Cognitive walkthroughs are best applied by the developers themselves, whereas guidelines have the additional advantage that they can be applied by others on the development team, e.g., technical writers.

Much time and money can be saved if problems are identified and corrected early in the development process. For the exact techniques used in this experiment, only cognitive walkthroughs could be effectively applied to early prototypes. However, there are (significantly different) variations of all of the other techniques that might have an impact on earlier stages of development; further research is needed to determine the actual effectiveness of those variants.

The cost of the analysis is another important consideration. We have already discussed the relative amounts of time the various techniques take, under the benefit/cost analysis. Usability testing is the only one of these techniques that requires significant additional out-of-pocket costs — to pay subjects, to equip a testing lab, for videotapes, etc. However, there are other types of costs for all the techniques. Fitting time for the analysis into a schedule is an important cost. Whether it is better to save money by having developers do the evaluation, or whether it is more effective to have someone who is not already overburdened with critical tasks, will depend on the particular project. In addition, the overhead of learning a new technique should not be minimized. Finally, a cost we did not directly consider is the cost of translating the problems found into changes to be made in the interface. There are significant costs to communication between developers and evaluators; aspects of the techniques that either increase the chances that the results of the evaluation will be taken seriously or that decrease the communication costs (by making the developers and the evaluators one and the same) can contribute positively to the eventual quality of the user interface and lower the cost of achieving that quality.

The most important conclusion to draw from these results is that no one evaluation technique is sufficient to find all the usability problems in a user interface of realistic complexity. None of the techniques found all the relatively serious problems in this interface – in fact, the likelihood that any particular problem would also be found by a different technique was about 10%. Thus, the more different evaluation techniques that can be applied, given the constraints of the development schedule, the more problems overall that will be found. We are continuing to attempt to characterize the differences in the problems found by the different techniques; a better understanding of the types of problems that each is best at identifying would be very helpful for persons trying to choose among the different options.

References

1. Brown, C. M. *Human-computer interface design guidelines.* Norwood, N. J.: Ablex Publishing Corporation, 1988.

2. Hewlett-Packard Company, Corporate Human Factors Engineering. *SoftGuide: Guidelines for usable software.* 1990.

3. Jeffries, R., Miller, J. R., Wharton, C., and Uyeda, K. M. User interface evaluation in the real world: A comparison of four techniques. In *Proceedings of CHI'91* (ACM Computer Human Interaction), New Orleans, April 28–May 3, 1991.

4. Lewis, C. and Polson, P. Theory-based design for easily-learned interfaces. *Human-Computer Interaction*, 1990, *5*, 2/3, 191–220.

5. Lewis, C., Polson, P., Wharton, C., and Rieman, J. Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In *Proceedings of CHI'90* (ACM Computer Human Interaction), Seattle, Washington, April 1–5, 1990.

6. Nielsen, J. and Molich, R. Heuristic evaluation of user interfaces. In *Proceedings of CHI'90* (ACM Computer Human Interaction), Seattle, Washington, April 1–5, 1990.

7. Smith, S. L. and Mosier, J. N. *Guidelines for designing user interface software.* Report MTR-10090, The MITRE Corporation, Bedford, Massachusetts, 1986.

User Interface Evaluation in the Real World: A Comparison of Four Techniques

Robin Jeffries

Hewlett-Packard Laboratories

1

HEWLETT PACKARD

Problems with user interface evaluation

- **It doesn't find all the usability problems**
- **It happens late in the development cycle**
- **It's expensive in time and resources**

2

HEWLETT PACKARD

New evaluation techniques

Cognitive walkthroughs

- Polson, Lewis, Wharton, & Riemann (1990)

Guidelines

- Smith & Mosier (1986)
- Brown (1988)
- SoftGuide (1990)

3

HEWLETT PACKARD

The methodologies

Cognitive walkthroughs

Guidelines: HP SoftGuide

Heuristic evaluation: by multiple experienced UI professionals

Usability testing: actual product usability test

4

HEWLETT PACKARD

The evaluators

Cognitive walkthroughs

- 3 software engineers

Guidelines

- 3 software engineers

Heuristic evaluation

- 4 individuals with HCI backgrounds

Usability testing

- 1 human factors professional
- 6 users tested; all experienced with PCs but not Unix

5

HEWLETT PACKARD

Total problems found

	Heur Eval	Usa-bility	Guide-lines	Cog Walk	Total
Core	121	32	35	35	223
Underlying system	15	3	3	0	21
Evaluator error	7	0	0	3	10
Non-repeatable	6	3	0	2	11
Other	3	0	0	0	3
Total	152	38	38	40	268
Core, no dups	105	31	35	35	206

6

HEWLETT PACKARD

How core problems were found

	Heur Eval	Usa-bility	Guide-lines	Cog Walk
via technique	105 (100%)	30 (97%)	13 (37%)	30 (86%)
side effect	--	1 (3%)	8 (23%)	5 (14%)
prior experience	--	0	12 (34%)	0

7

HEWLETT PACKARD

Severity analysis

Restricted to core problems

Seven raters

- **Four usability experts**
- **Three HCI-knowledgeable software engineers**

1 — 9 scale, anchored

Told to consider: impact, frequency, number of users impacted

8

HEWLETT PACKARD

Severity results

	Heur Eval	Usa-bility	Guide-lines	Cog Walk
Mean problem severity				
	3.59	4.15	3.61	3.44
Number found by severity				
most severe	28	18	12	9
least severe	52	2	11	10

9

HEWLETT PACKARD

Benefit/cost analysis

	Heur Eval	Usa-bility	Guide-lines	Cog Walk
	Benefit: Sum of severity scores			
linear	433	133	130	120
exponential	355	175	205	81
	Cost: Time spent on analysis (person-hours)			
analysis time	20	199	17	27
technique training	--	--	5	10
HP-VUE training	15	--	64	6
	Range of benefit/cost ratios: severity/time			
	12-10	1	12-2	4-2

10

HEWLETT PACKARD

Advantages and disadvantages

Heuristic evaluation

+ Most cost-effective
+ Finds most problems in all categories
- Requires UI expertise
- Requires multiple experts

Usability testing

+ Finds serious and recurring problems
- Requires UI expertise
- Expensive
- Misses consistency problems

11

HEWLETT PACKARD

Advantages and disadvantages

Guidelines

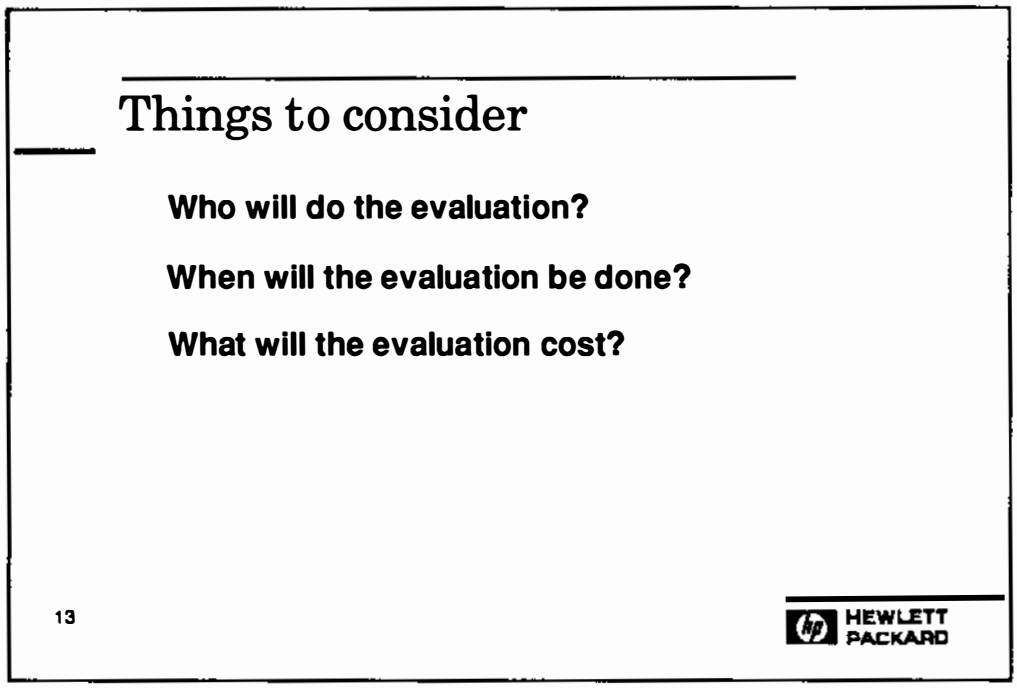
+ Usable by developers
+ Finds recurring and general problems
• Moderate cost, moderate benefit

Cognitive walkthroughs

+ Helps define users' goals and assumptions
• Usable by developers, but needs task definition methodology
• Moderate cost, moderate benefit
- Tedious

12

HEWLETT PACKARD


Things to consider
Who will do the evaluation?
When will the evaluation be done?
What will the evaluation cost?
13
HEWLETT PACKARD

Compiler Support for Program Testing on MIMD Architectures *

R. A. DeMillo
E. W. Krauser
A. P. Mathur

Software Engineering Research Center
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-2004

Abstract

Traditionally, available compilers have supported only two simple testing techniques, namely, statement and branch coverage. However, during compilation, sufficient syntactic and semantic information is available to provide support for more sophisticated testing techniques. This paper presents a method to efficiently support program mutation via information and code generated at compile-time. The method is also applicable to other white-box testing methods such as data flow testing.

Mutation requires information attained by modifying the internal execution behavior of a program under test. Since development of an adequate test data requires repeated program executions, extensive testing may be expensive and time-consuming. Support for program mutation via a compiler-based approach, as opposed to a traditional interpreter or separate compilation based approach, is expected to afford a significant increase in the execution speed and cost-effectiveness of a mutation-based software test. Past research indicates that this approach is perhaps essential for the efficient application of hypercube machines (e.g., the Ncube/2) to mutation testing.

Keywords: Branch coverage, compilers, data flow testing, debugging, incremental program modification, profiling, program mutation, software testing, statement coverage.

1 Introduction

This work is concerned with compiler-integrated support for *program mutation* [2]. Mutation requires information attained by modifying the internal execution behavior of a program P under test. Faults are injected into P that are intended to model simple errors possibly introduced by programmers using a specific programming language L. Each fault, or mutation, is obtained by a single point, syntactically correct change to the original program P. Examples of faults induced include, variable replacement, e.g. $x = x + y$ replaced by $x = y + y$ and operator replacement, e.g. $x = x + 1$ replaced by $x = x - 1$. Each program resulting from such a change to P is called a *mutant* program. The goal of a mutation tester is to select test data such that the output of P is distinguished from the output of all mutant programs which are not equivalent to P. Such

*This research was supported, in part, by a grant from the Software Engineering Research Center at Purdue University, a National Science Foundation Industry/University Cooperative Research Center (NSF Grant No. ECD-8913133).

test data is deemed to be 'mutation adequate' and regarded as offering very strong evidence that a well chosen set of simple errors do not exist in P. More importantly, an established theoretical framework and large body of empirical evidence suggest that mutation adequate test data also reveals a much larger set of complex errors in P [1].

Tools currently exist that support program mutation via an interpreter-based execution environment [5]. However, they execute mutant programs much slower than if the mutants were to consist of native machine code. Therefore, testing tools based on current approach are not attractive as a means to test any reasonably large program (e.g. consisting of more than 10,000 lines of code). We present a method that permits efficient support for program mutation via information and code generated at compile-time, hence, we call this a compiler-integrated testing approach, hereafter referred to as CIT.

The remainder of this paper is organized as follows. The next section examines existing tools and past research into mutation to motivate the need for CIT for mutation testing. The need for for such an approach in a parallel environment is also stressed in this section. Section 3 describes patch generation, a patch being a sequence of object instructions. The organization and representation of patches is explained in Section 4. The process by which the patch file ism constructed is described in Section 7. Applicability of CIT to other testing methods is discussed in Section 8. Section 9 summarizes our work and outlines plans for future work.

2 Motivation

To understand the need for CIT for mutation, consider the approach of *Mothra* [5], an existing mutation-based FORTRAN 77 testing environment (see Figure 1). The program P under test is input to the Mothra system and translated to an internal representation P_{int}. Mutants of P are generated by inducing changes into P_{int} required to model each desired fault. Finally, each mutant is first constructed by applying the appropriate changes, or edits, to P_{int} and then interpretively executed on test data until either, (1) its output is distinguished from that of P, or, (2) it has been executed on the entire set of test data. An undistinguished mutant program is considered to be *live*, while a distinguished mutant is considered to be *dead* and a mutant that cannot be distinguished is considered to be *equivalent*. Once so labeled, dead and equivalent mutants no longer participate in the software test. The test data set is continuously augmented until either, (1) all mutants have been labeled dead or equivalent, or, (2) a threshold percentage of dead mutants has been reached This threshold, known as a mutation score, is computed as $\frac{\#\ dead\ mutants}{\#\ live\ mutants\ -\ \#\ equivalent\ mutants}$.

Empirical evidence has shown mutation to be an effective testing technique [1]. However, one can see that any mutation-based testing environment must be faced with the cost of executing a large number of mutant programs [1]. Consider *trityp* [14], a 29 line triangle identification program written in FORTRAN 77. When mutated, *trityp* generates a total of 970 mutants programs, each of of which must be executed in an effort to distinguish their output from that of the original. One automatic test data generator produced 420 test cases for the *trityp* program [14]. In the worst case, all mutants and the original program must be executed with respect to all test data, a total of 407,820 program executions! In general, a typical mutation software test does not realize this worst case scenario. For example, many mutants are very unstable and are easily distinguished. As mutants are killed or asserted to be equivalent, there is no need to execute them with respect to subsequent test data. This often significantly reduces the required number of program executions. However, if even 10% of these executions need be managed, the resulting computational load

[1] The computational complexity of mutation is polynomial in the number of distinct identifiers referenced in the program. A mathematical analysis of this complexity appears in [2].

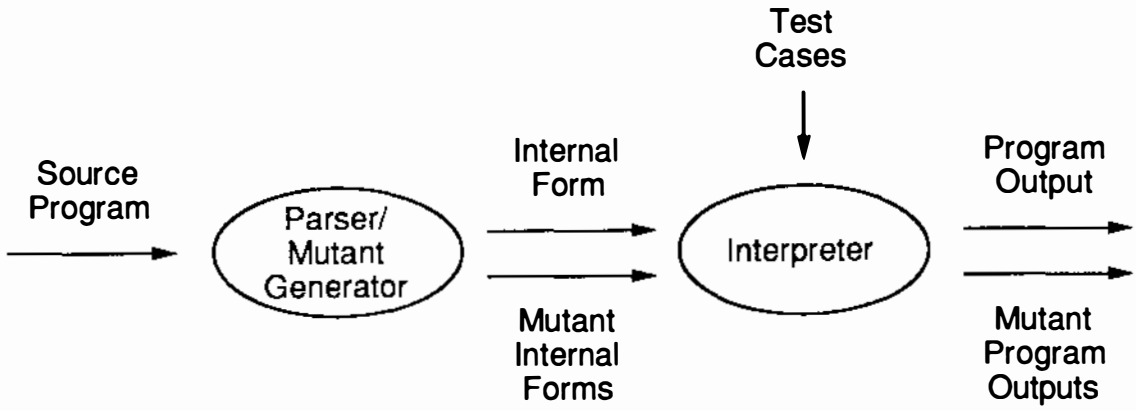


Figure 1: An interpreter-based approach to mutation testing.

becomes time consuming and costly on traditional sequential machines.

In addition to being strapped with a CPU-intensive task, Mothra suffers from the inherent slow execution speed of an interpretive execution environment. A straightforward solution to improve slow execution speed is to replace interpretive execution by execution of compiled code, as depicted in Figure 2. Programs execute faster and retain their original operational behavior (e.g., timing characteristics) while executing in their intended operational environment. However, this approach has a disadvantage. In replacing interpretive execution by machine execution, the number of required compilations has increased. That is, before each mutant can be executed, it must be transformed into executable object code via the application of a compiler, assembler, and linker/loader. This introduces significant overhead into the testing process.

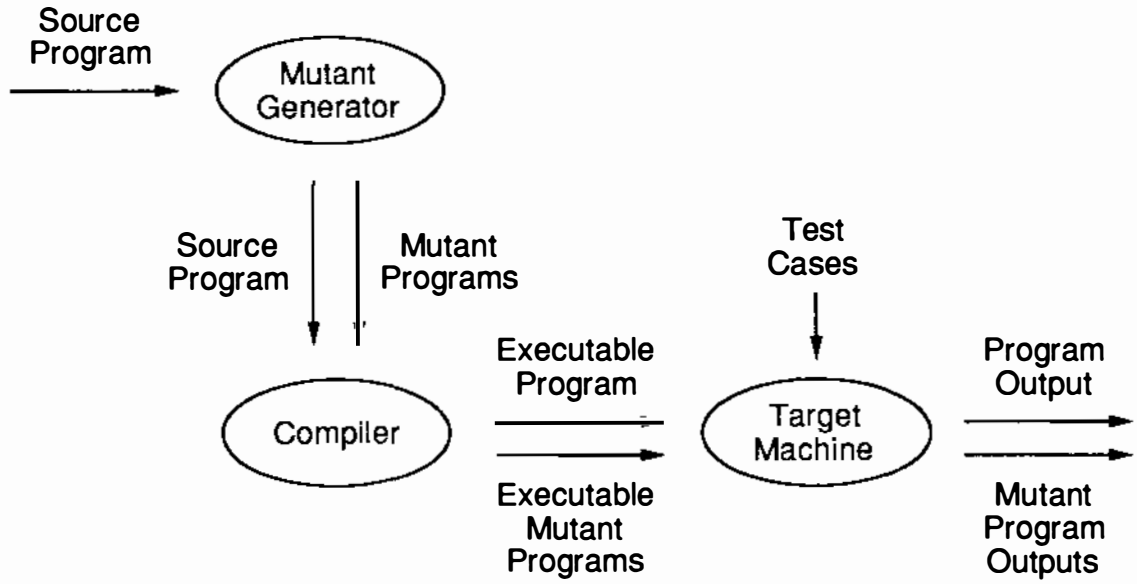


Figure 2: A separate compilation approach to program mutation.

Furthermore, compiling each mutant individually suffers from redundancy because, by definition, each mutant is very similar to the program under test. This observation leads directly to CIT as illustrated in Figure 3. The program P under test is input to the compiler which generates

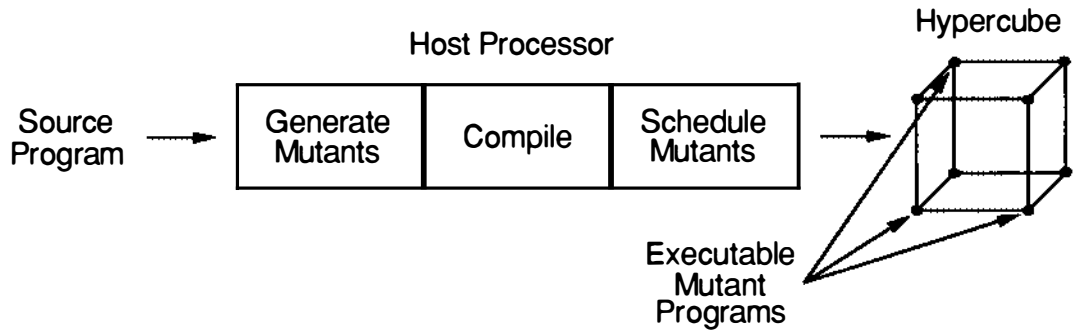


Figure 4: The organization of mutant program execution in PMothra.

Table 1: The symbols relevant to the PMothra experimental model.

Symbol	Description
λ_c	Mean compilation time of a mutant program.
l	Time to load a mutant program for execution.
λ_t	Mean mutant execution time.
λ_o	Mean mutant output comparison time.
t	$\lambda_t + \lambda_o$
s	Mean service time for a test case.
η	Mean number of test cases required to kill a mutant.
N_p	Number of processors dedicated to mutant program execution.

improve processor utilization for the problematic class of programs identified by Choi. The result is intended to provide a platform for the construction of a new PMothra tool that, as indicated by experimental results, will provide a significant improvement in the cost-effectiveness of program mutation. Below we discuss the specifics of Choi's results in detail and the nature of the compiler support required to effectively apply MIMD architectures to program mutation.

2.1.1 Choi's Results

Consider the separate compilation approach to mutant program execution depicted in Figure2. Assume that the target machine is multi-node hypercube. Then a mutation-based testing tool can be constructed to run on a host processor and control the execution of mutant programs on the hypercube. Each mutant is generated by a source level edit to the program under test, compiled, and scheduled for execution by the host processor. When its scheduled node becomes available on the hypercube, the mutant is sent to the hypercube to be loaded, executed and compared with the output of the program under test. Once scheduled, mutants remain at their assigned node and continue to execute until killed or all test data has been exhausted. This is an attempt to maintain high processor utilization and is consistent with the basic design of PMothra. Although, Choi has proposed a design for a somewhat more elaborate PMothra system with superior capabilities to respond to the dynamic scheduling requirements of such a system. Figure4 depicts the PMothra scenario while Table1 defines a set of symbols to be referenced in the following discussion.

Choi reports near linear speedups for programs whose mutants require relatively long execution times as compared to the time it takes to compile and load them for execution. This result,

depicted in [3], makes perfect sense and is explained as follows. Assume the first mutant $\mathcal{Q}_0$ has been compiled, loaded, and is about to commence execution on node n_0. Based on the definitions in Table1, on the average this requires $\lambda_c + l$ time, the sum of the mean time to compile and load a mutant. At this point, the other $(N_p - 1)$ nodes remain idle, awaiting a mutant for execution. The mean time node n_0 will remain busy executing mutant $\mathcal{Q}_0$ is $(t + s) \times \eta$, the sum of the mean execution and service times multiplied by the mean number of executions before the mutant is killed. In order to attain high processor utilization by servicing the $(N_p - 1)$ idle nodes, $(N_p - 1)$ mutants must be readied for execution before mutant $\mathcal{Q}_0$ is killed. Also, an additional mutant must be ready to replace $\mathcal{Q}_0$ at node n_0. Therefore, if speedup is to be attained, the mean compile-time and mean load-time for mutant programs are subject to the following constraint:

$$\lambda_c + l \leq \frac{(t + s) \times \eta}{N_p} \tag{1}$$

If this constraint is not met, processor utilization falls off because there are idle nodes waiting for mutant programs to be readied for execution. As shown in [3], this results in poor speedup and cost-effectiveness.

Unfortunately, this is the case for a large class of programs. Consider TEX, the text formatting program developed by Knuth[10, 11]. TEX consists of about 5,000 mutatable source lines of Pascal. At most installations, the executable version of this Pascalprogram is "built" by first preprocessing the program into C code and then compiling it normally, a process that takes approximately 5 minutes on a SUN SPARCstation1. Yet, a typical test case for TEX may only require a few seconds to execute. Therefore, the mean total execution time for what is potentially a very large number of mutants is likely to be less than the typical build-time. In addition, the build-time does not include the overhead associated with communicating all of these mutants to their destination nodes and loading them for execution once they arrive.

A final discouraging conclusion to be drawn from Choi's results is that attempting to exploit the modularity of a program and conduct unit testing, a typical scheme to make a software test manageable, may not help. That is, Equation1 may well not hold in a mutation-based testing environment that supports unit testing because, although the mean compilation and load time may be reduced, so is the mean execution time.

3 Solution Techniques

The results presented by Choi lead directly to the CIT as illustrated in Figure3. The main goal is to ensure that Equation1 is always satisfied. This is achieved by a two-phase solution: the generation of *program patches* at compile-time, coupled with subsequent selective patch application when the software is under test. Each of these phases is briefly described below.

3.1 Patch Generation in CIT

The program under test is input to the compiler which generates two outputs: an executable object code image and a set of *program patches*. Each program patch consists of one or more code sequences and a corresponding set of editing operations that direct how the code sequences are to be applied to the compiler's resultant executable. When a program patch is applied to the executable, the result is a new executable with possibly different semantics. The focus of this research has been to construct program patches such that when applied, the resulting executable is that of mutant program. The approach effectively folds the compilation of all mutant programs

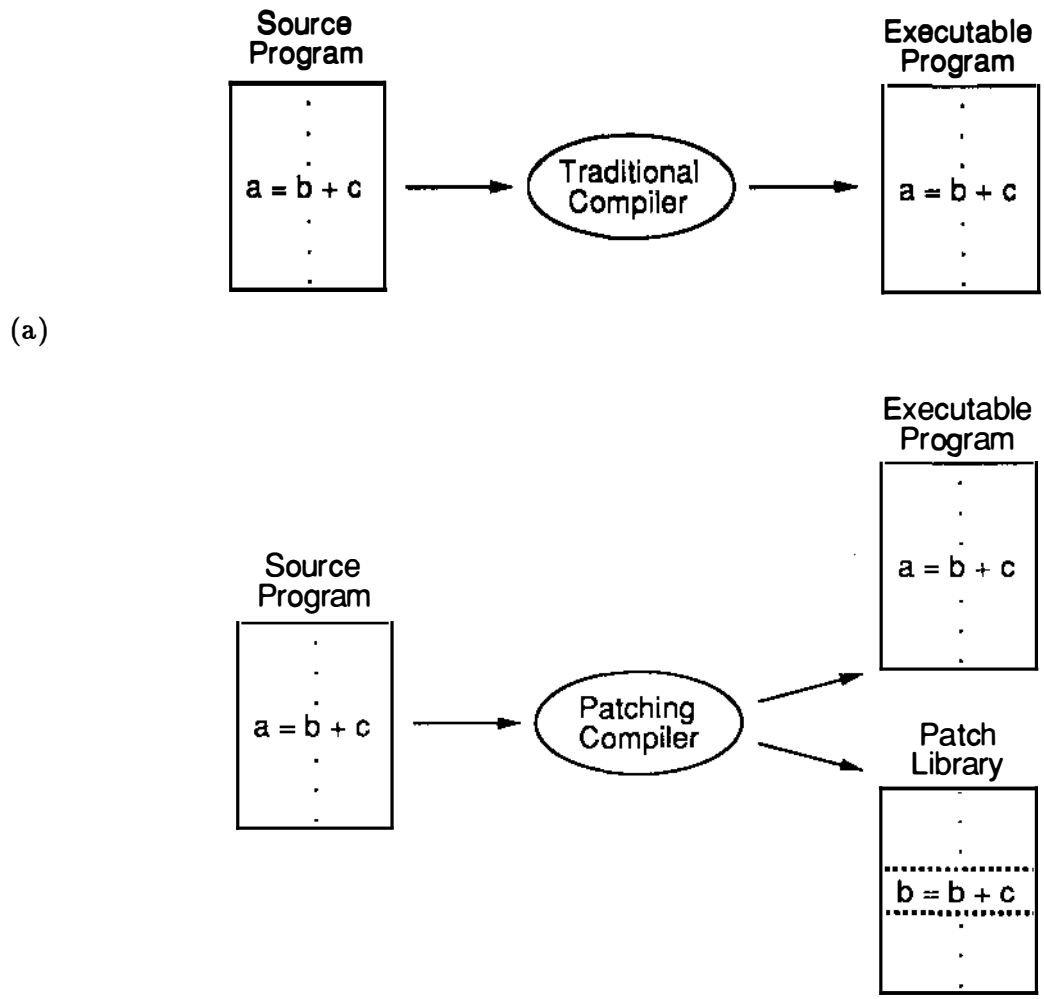


Figure 5: (a) Traditional compilation versus (b) compilation in CIT.

into one compilation, encapsulating the "diffs" of each mutant and the program under test in the form of a patch. That is, only the underlying machine instructions that differ between the original and its mutant programs are compiled. Figure 5 depicts the compile-time program patch generation process.

3.2 Selective Program Code Patching

During testing, when mutants are to be executed, program patches are selectively applied to the compiler's resultant executable in order to obtain each mutant program. Each time a patch is applied to the program under test, a mutant program is obtained, as is depicted in Figure 6.

The application of program patches occurs via the long-used technique of *code patching*. An arbitrary sequence of instructions are appended to the object image of an executable program. Then, jump instructions are overwritten into the instruction stream of the executable in order to redirect its thread of execution at run-time and execute the instructions appended to the end of its image. In this way, the desired sequence of instructions appear to have been inserted directly into the instruction stream of the executable. When each program patch is generated such that it

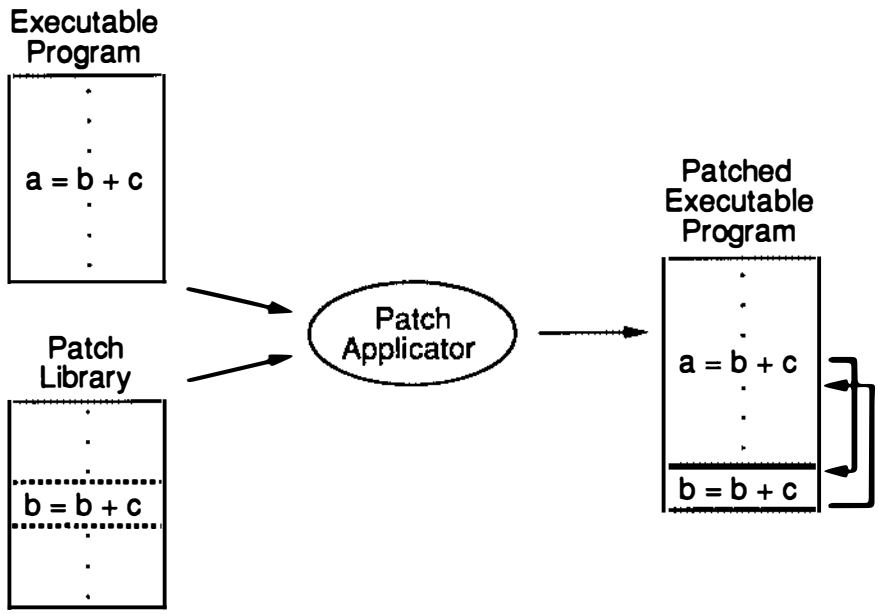


Figure 6: Program patch application to construct mutant programs during the adequacy measurement phase.

encapsulates the differences between the executable under test and its mutants, then the application of each patch yields an executable mutant program. Call these patches *mutant patches.*

The application of mutant patches to the compiler's resultant executable is an efficient process with little overhead. This is because effort has been expended to make sure that all references within a mutant patch are resolved with respect to the symbol table of the program under test *before* application. This precludes the need for dynamic linking and makes patch application equivalent to a simple set of bit stream edits on the executable file.

4 Patch Organization & Representation

Each source level program mutation requires that one or more new sequences of machine instructions be installed in the executable image $\mathcal{P}$ of a program. Therefore, each program patch consists of a sequence of parameterized operations on an executable program image. This sequence has the form:

$$\langle OP_1, \ldots, OP_n \rangle,$$

where OP_i is either a *data directive* or an *editing operator* associated with a specific *patch instruction sequence*, denoted by PIS_i.

During compilation, as each patch M is generated, its instruction sequences do not become part of those for $\mathcal{P}$. Rather, each PIS_i associated with M is stored in a patch library, which is designed to provide selective access to M during program testing. During a subsequent test data adequacy measurement phase of program mutation, the sequence of operations within M are interpreted by a *patch applicator* to obtain an executable mutant program $\mathcal{P}_M$. Directives to the patch applicator allow data to be associated with each PIS_i in M, while editing operators install each PIS_i for the program mutation induced by M within the instruction stream of $\mathcal{P}$. This is achieved by overwriting appropriate locations within $\mathcal{P}$ with jump instructions. The effect is to redirect the run-time thread of execution in $\mathcal{P}$ and include each new PIS_i in M, yielding the behavior of $\mathcal{P}_M$.

Each PIS_i applied to $\mathcal{P}$ is installed by the patch applicator in the form of a *patch instruction record*, denoted by PIR_i. The set of all PIR_i installed in $\mathcal{P}$, along with any static data they require comprise the *patch segment*, denoted by *PatchSeg*. *PatchSeg* is a contiguous region of space within $\mathcal{P}$ whose size and location are determined by the patch applicator. *PatchSeg* is allocated prior to the installation of M and must be of sufficient size to store any static data and each PIS_i for M. In addition, *PatchSeg* must be installed in a location that does not invalidate any relative, relocatable, or symbolic addresses within the instructions of $\mathcal{P}$. Once installed, the patch applicator initializes the global variable *PatchSegPtr* within $\mathcal{P}$ to the address at which *PatchSeg* was installed within $\mathcal{P}$.

5 Patch Instruction Sequences

Each sequence of operations for a given patch M contains one *primary instruction sequence* and zero or more *secondary instruction sequences.* A primary instruction sequence is that which arises directly from the parse tree transformation that models a given mutation within the source program. Secondary instruction sequences are comprised of additional instructions required to ensure the correctness of the primary instruction sequence. Each secondary instruction sequence must be executed at a location within $\mathcal{P}$ other than that where the primary instruction sequence occurs, and therefore cannot be included within the primary instruction sequence. Secondary instruction sequences are generated in order to perform the following tasks:

- Allocate and deallocate space for *temporary variables* within the activation record of a function f to which M has been applied;
- Reset *loop-counter variables* within M when the iterative construct to which M has been applied is exited.

6 Associating Patches with Data

Each patch instruction sequence may optionally be associated with *static* and/or *dynamic* data. Static data required by any patch is allocated and initialized to zero by the patch applicator within a contiguous region at the beginning of *PatchSeg*. Such data is accessible to any patch applied to $\mathcal{P}$. Dynamic data for any patch M is allocated within the activation record of the function f to which M is applied. Such data cannot solely be allocated by the patch applicator. Rather, the calling and return sequences for f are patched to respectively allocate and deallocate the extra temporary variables required for M at run-time. This necessitates that the compiler generate at least three instruction sequences for any patch requiring dynamic data, one to affect the mutation and two to allocate and deallocate temporary variables. Dynamic patch data is accessible to the instruction sequences of any patch applied to f, but not by any instruction sequences that are applied outside of f.

6.1 Data Requirements

Dynamic data may be required by any patch corresponding to a *operator mutation* or *variable mutation* because evaluation of a mutated expression may induce the need for compiler-generated temporary variables not required in evaluating the original expression. This situation may arise for two reasons:

- *Type conversions.* Types are ignored during *Scalar Variable Replacement.* Therefore, replacement of variable v by variable v' having a higher-order type may necessitate one or more type conversions within an expression that required no such conversions in the original program. Likewise, *operator replacement mutations* may, depending on the operands, cause conversion of the value of an operand where no such type conversion was required in the original program. Each type conversion requires a new temporary variable within the intermediate representation of the mutated expression.

- *Computation of offsets.* All *variable mutations* can necessitate the computation of offsets not required within the original program. For instance, let v be a variable and s be a structure defined as:

  $$\texttt{int } v; \quad \texttt{struct } \{\texttt{int } u; \ \texttt{int } v;\} \ s;$$

  Then replacing variable v by $s.v$ requires a new subexpression to compute the address of $s.v$ with respect to the address of s. Each such subexpression requires a new temporary variable within the intermediate representation of the mutated expression.

Temporary variables store new intermediate states of computation that arise from a program mutation and for which no unused registers or temporary variables are currently available. For recursive languages like C such data must be allocated on the run-time stack because several invocations of a patched construct may be active at the same time, each requiring its own temporary variables. For a non-recursive language, all temporary variables may be allocated statically within *PatchSeg*.

Patches corresponding to *coverage mutants* [2] require both static and dynamic data. The static data required is comprised of data structures that record the satisfaction of coverage criteria detected by a given patch and may be shared by all such patches for a program. For example, the patch for a *Statement Trap* mutant need only record that a given statement has been executed by setting a bit in an static array, where each bit represents a statement in the program. The dynamic data required is comprised of temporary variables into which expressions are evaluated or loop-counter variables that record which iteration of a given looping-construct is currently active.

7 The Making of the Patch File

Because compilation is a translation process, at some point in time, analogs of each patch must exist at the parse tree, intermediate code, assembly code, and object code levels. In this section, we provide and overview of how this process is managed, resulting in the construction of an object patch file. The remainder of the chapter then discusses in detail precisely how patches are managed and refined during each phase of compilation, and how they are tailored to support C program mutations.

7.1 File & Unit Entries

The patch database is organized in a manner that allows program patches to be accessed by file, subprogram unit (i.e., function), type, as well as the line and column in which they occur in the source program. Each patch also permits linkage to an external, application-specific database via a primary key stored within the patch. The patch applicator locates the sequences of instructions corresponding to each patch by looking up the labels that delimit each patch in the symbol table

[2] A *coverage mutant* is a *statement mutation*, such as the *Statement Trap* operator, that detects the satisfaction of various coverage criteria during program execution.

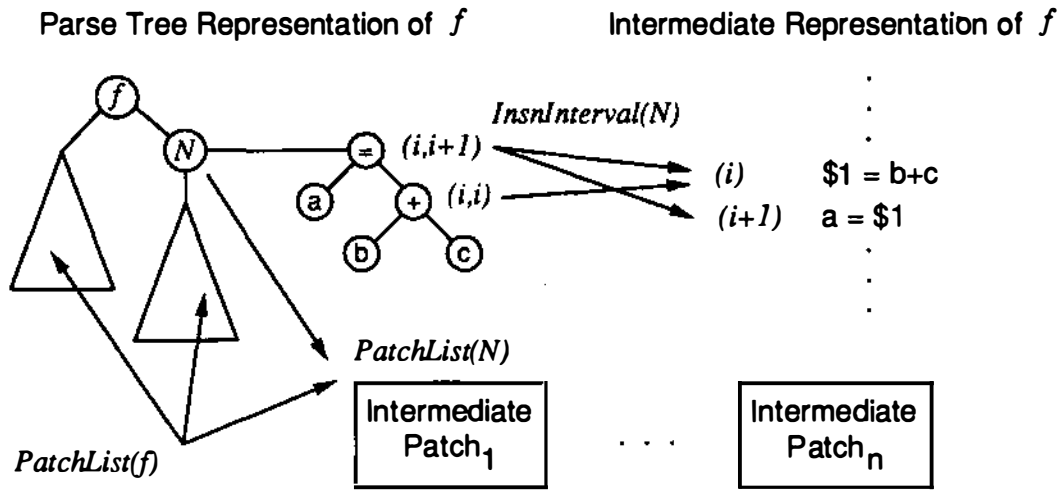


Figure 7: Organization of what has been produced by a patch generating compiler following the intermediate code generation phase.

of the patch library. Patch instruction sequences are then accessed by extracting that region of the patch library indicated by the resulting address interval.

Entries in the patch database are first allocated on a function-by-function basis during parsing. As each source code file within a program is parsed, one *file entry*, as well as one *unit entry* per function defined in the file, is created. The file entry maintains pointers delimiting the sequence of contiguous unit entries corresponding to its underlying source file, while each unit entry stores a pointer to its enveloping file entry. This organization provides for selective patch access in the presence of separate compilation, where a set of patch files, each resulting from the compilation of a distinct source file, are merged to yield a composite patch file.

7.2 Patch Entries

During intermediate code generation, as the set of mutants for the current function f are identified, one *patch entry* is created per mutant generated. Each such patch entry in the patch database is assigned a type, source location, and pointer recording its enveloping unit entry. As each sequence of patch operations and its corresponding sequences of intermediate code instructions, are generated, they are assigned to the patch entry. Each patch entry is then associated with both a global list of patches for f, denoted by $PatchList(f)$, and the parse tree node N, where N roots the domain entity for which the patch was generated. In addition, the correspondence between the sequence of intermediate code instructions generated for each syntactic entity in f and the parse tree is maintained by associating a tuple $InsnInterval(N)$ with each parse tree node N. This organization results in a mapping from intermediate code instructions in f to domain entities and their corresponding patches in the parse tree. Finally, once all patches for f have been generated, the unit entry created for f during parsing is updated to store pointers delimiting its corresponding sequence of contiguous patch entries created during patch generation. Figure 7 illustrates the organization of what has been produced by a patch generating compiler following the intermediate code generation phase.

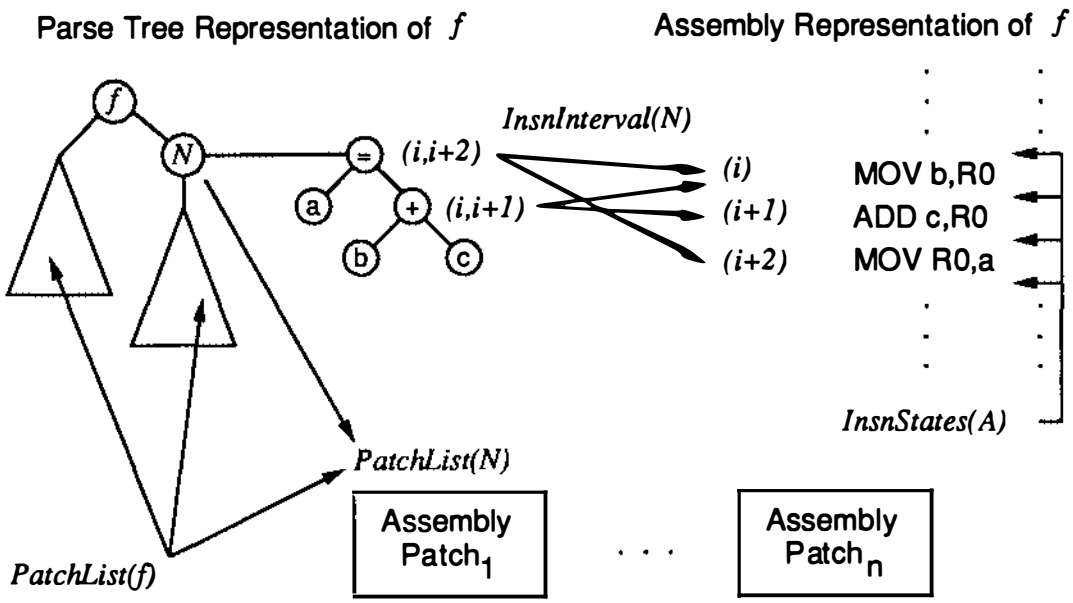


Figure 8: Organization of what has been produced by a patch generating compiler following the assembly code generation phase.

7.3 Assembly Code Patches

During assembly code generation, the assembly representation of f is generated first, although the resulting instructions are not immediately emitted to the target assembly file. As each assembly instruction A is produced, the state of the machine, as perceived by the compiler, before and after generation of A is recorded in $InsnStates(A)$. The correspondence between the assembly code instructions generated for each syntactic entity in f and the parse tree is maintained by refining the tuple $InsnInterval(N)$, produced during intermediate code generation. Following translation of the last intermediate code instruction for the subtree rooted at N, this tuple is updated to reflect the sequences of instructions within the assembly representation of f, rather than within the intermediate representation of f. Once the entire assembly representation of f is available and its mapping onto domain entities within the parse tree known, $PatchList(f)$ is traversed and each patch produced during intermediate code generation is translated to assembly instructions, in turn.

Let domain entity E be rooted at node N where $InsnInterval(N) = (A_i, A_f)$, for initial and final assembly code instructions A_i and A_f, respectively. Generating an assembly patch instruction sequence PIS_i for patch M replacing domain entity E requires restoring the machine state perceived by the compiler to that before A_i was generated, generating PIS_i, and generating *state restoring instructions* to restore the machine state prior to execution of A_f. All assembly instruction sequences generated for M are then delimited by patch instruction labels and emitted to the patch library. Similarly, patch target labels are generated and inserted into the assembly representation for f at the locations to which the editing operators of M are to be applied. Finally, each editing operator within M is updated to include the patch target and instruction labels that are now known and may be bound to patch M. Figure8 illustrates the organization of what has been produced by a patch generating compiler following the assembly code generation phase.

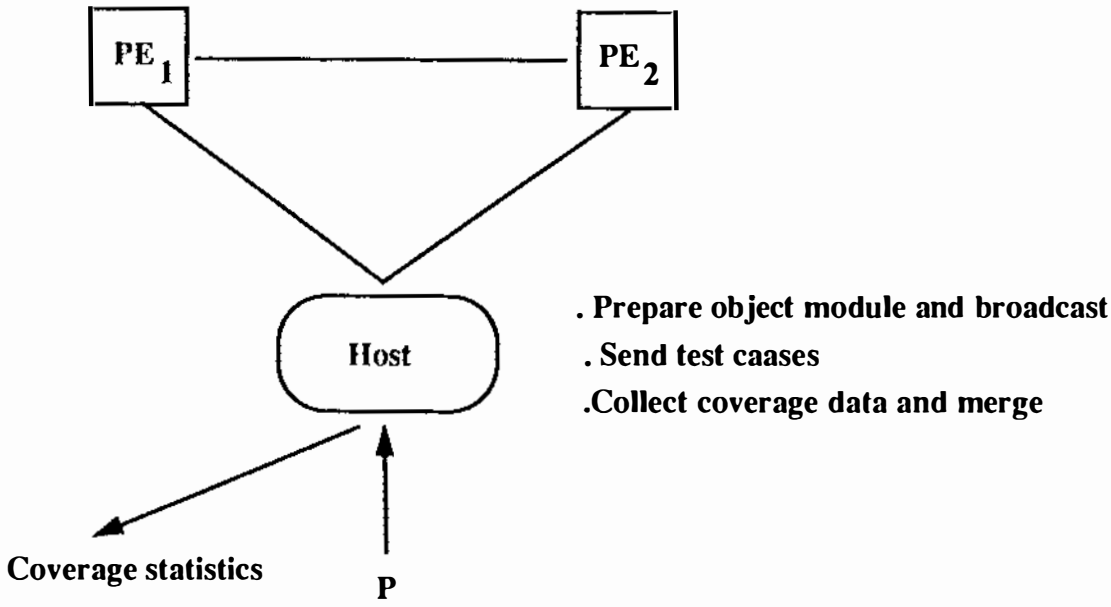


Figure 9: Data flow testing on an MIMD machine.

8 Applicability to Other Testing Methods

Though CIT is described in this paper as it is applied to mutation testing, it is indeed applicable to other testing methods also. In fact we claim that any testing method that requires a modification to the program being tested can benefit from CIT. To illustrate how, below we provide an example of applying CIT to improve the performance of data flow testing.

Tools that support data flow testing, e.g. ATAC [7], instrument the program under test. When executed, the instrumented program records coverage data, e.g decision or p-use coverage. However, the instrumentation causes a reduction in the speed of program execution. We have observed speed reductions ranging from two to thirty times when using ATAC. The exact amount of reduction depends on several characteristics of the program being tested, e.g. I/O or CPU bound. It is, however, possible to use an MIMD machine, such as Ncube's Ncube/2, to speedup the execution of the instrumented program.

Figure 9 exemplifies how the CIT approach can be used to organize the instrumentation of the program P under test to obtain data flow coverage. Consider a machine with two processing elements denoted by PE_1 and PE_2 in the figure. The PE's can communicate with a host as well as amongst themselves.

Let P consist of four functions f_1, f_2, f_3, and f_4. If a data flow testing tool for sequential

machines, such as ATAC, is used to instrument P then it would instrument all the four functions to obtain data flow coverage. Let P' denote such an instrumented program. It is now possible to broadcast the object version P'_o of P' to each of the PE's for execution on different test cases. Note that due to instrumentation, each P'_o will execute significantly slower than P_o, the object version of P.

Let us carry out a simplified time analysis of this approach. If T denotes the time to execute P and δ the instrumentation overhead associated with each function in P, then the total time to execute P' on 2 PE's and 2 test cases is $T + 4\delta$. Note that the execution of P' on each test case can be carried out in parallel. Here we have assumed that each function in P is executed exactly once on each test case. Clearly, this approach yields a speed gain of 2 over the approach that uses a sequential machine.

The instrumentation of P can be carried out by a separate testing tool or by the compiler itself. Using a compiler that has such instrumentation capability avoids having to build or buy a separate tool. Further, a separate tool would perform activities such as parsing and semantic analysis to be able to do the instrumentation. The compiler in any case performs these activities. Hence, using a compiler for this purpose avoids having to perform program analysis twice. Note that the advantages of the CIT approach we mention here are applicable to both sequential and parallel machines. This, however, is not true for mutation testing as was explained in Section 2.1.1.

9 Summary and Future Work

In this paper we have presented a method to integrate support for program mutation and coverage based testing methods into a compiler. As program mutations can be modeled as localized program edits, this method is believed to be general enough to also allow support for other software development, testing, and maintenance activities such as debugging, data flow testing, incremental program modification, and profiling. Finally, previous research suggests that this approach appears to be essential for the efficient application of hypercube machines to program mutation. Such an approach may be the only feasible method of testing large commercial software systems in a mutation-based testing environment.

We are currently building a test environment based on CIT. At the heart of this environment is the GNU C compiler which has been modified to generate patches as described in this paper. In addition to the compiler, the environment consists of a patch applicator, test case editor and manager, test display manager, and a data base for storing the status of the software test. The modified compiler is also being interfaced with PMothra/ [4], a tool for scheduling mutants on the Ncube/2 hypercube.

It is believed that CIT will provide a significant increase in the efficiency of several existing testing tools and allow program mutation to be effectively employed to test commercial software systems. This hypothesis has yet to be empirically substantiated. However the approach seems to be innately appealing to the software tester because it also enhances the reliability of a software test. The program under test retains much of its original operational behavior (e.g., timing characteristics, while executing in its intended operational environment.

References

[1] T. A. Budd. *Mutation Analysis of Program Test Data.* PhD thesis, 1980.

[2] Timothy Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation Analysis. Technical report, Department of Computer Science, Yale University, April 1979. Subsequently released in expanded form as Technical Report GIT-ICS-79/08, Georgia Institute of Technology, 1979.

[3] B. Choi. *Software Testing Using High Performance Computers.* PhD thesis, 1990.

[4] B. Choi, A. P. Mathur, and B. Pattison. PMothra: Scheduling Mutants for Execution on a Hypercube. In *Proceedings of the Third Software Testing, Analysis, and Verification Symposium*, Key West, Florida, December 1989.

[5] R. A. DeMillo, D. S. Guindi, K. S. King, W. M. McCracken, and A. J. Offutt. An Extended Overview of the MOTHRA Mutation System. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff, Alberta, Canada, July 1988.

[6] R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Using the Hypercube for Reliable Testing of Large Software. Technical Report SERC-TR-24-P, Software Engineering. Research Center, Purdue University.

[7] J. R. Horgan and S. A. London. ATAC — Automatic Test Coverage Analysis for C Programs. Technical report, Bell Communications Research, July 1990.

[8] Inmos Corporation. *Occam 2 Manual*, 1985.

[9] Inmos Corporation. *Reference Manual and Product Data: Trasnputer*, 1985.

[10] D. E. Knuth. *TEX: The Program.* Addison-Wesley, Reading, MA, 1986.

[11] D. E. Knuth. *The TEX Book.* Addison-Wesley, Reading, MA, 1987.

[12] E. W. Krauser, A. P. Mathur, and V. Rego. High Performance Testing on SIMD Machines. *IEEE Transactions on Software Engineering*, 17(5), May 1991.

[13] Aditya P. Mathur and E. W. Krauser. Modeling Mutation On a Vector Processor. In *Proceedings of the 10th International Conference on Software Engineering*, Singapore, April 1988. Previously released as Technical Report GIT-SERC-87/07, Georgia Institute of Technology, 1987.

[14] A. J. Offutt. *Automatic Test Data Generation.* PhD thesis, Georgia Institute of Technology, 1988. Jeff's Thesis.

Factors That Affect Software Testability

Jeffrey M. Voas
Systems Architecture Branch
Information Systems Division
Mail Stop 478
NASA Langley Research Center
Hampton, VA 23665
(804) 864-8136
jmvoas@phoebus.larc.nasa.gov

Abstract: *Software faults that infrequently affect software's output are dangerous. When a software fault causes frequent software failures, testing is likely to reveal the fault before the software is released; when the fault remains undetected during testing, it can cause disaster after the software is installed. A technique for predicting whether a particular piece of software is likely to reveal faults within itself during testing is found in [Voas91b]. A piece of software that is likely to reveal faults within itself during testing is said to have high testability. A piece of software that is not likely to reveal faults within itself during testing is said to have low testability. It is preferable to design software with higher testabilities from the outset, i.e., create software with as high of a degree of testability as possible to avoid the problems of having undetected faults that are associated with low testability.*

Information loss is a phenomenon that occurs during program execution that increases the likelihood that a fault will remain undetected. In this paper, I identify two broad classes of information loss, define them, and suggest ways of predicting the potential for information loss to occur. We do this in order to decrease the likelihood that faults will remain undetected during testing.

Index Terms: Testability, Domain/Range Ratio (DRR), random black-box testing, information loss, information hiding, software design, specification metric.

Jeffrey Voas is working as a National Research Council resident research associate at the National Aeronautics and Space Administration's Langley Research Center. His research interests include software testing, studying data state error propagation, debugging techniques, and design techniques for improving software testability. Voas received a BS in computer engineering from Tulane University and a MS and PhD in computer science from the College of William and Mary.

Factors That Affect Software Testability

1 Introduction

This paper exposes factors that I have observed which affect program testabilities. *Testability* of a program is a prediction of the tendency for failures to be observed during random black-box testing when faults **are** present [VOAS91b]. A program is said to have *high testability* if it tends to expose faults during random black-box testing, producing failures for most of the inputs that execute a fault. A program has *low testability* if it tends to protect faults from detection during random black-box testing, producing correct output for most inputs that execute a fault. In this paper, I purposely avoid a formal definition for fault because of the difficulty that occurs when trying to uniquely identifying faults, and instead use the intuitive notion of the term fault.

Random black-box testing is a software testing strategy in which inputs are chosen at random consistent with a particular input distribution; during this selection process, the program is treated as a black-box and is never viewed as the inputs are chosen. An *input distribution* is the distribution of probabilities that elements of the domain are selected. Once inputs are selected, the program is then executed on these inputs and the outputs are compared against the *correct* outputs.

Sensitivity analysis [VOAS91b] is a dynamic method that has been developed for predicting program testabilities. One characteristic of a program that must be predicted before sensitivity analysis is performed is whether the program is likely to *propagate* data state errors (if they are created) during execution. Propagation analysis [VOAS91b, VOAS91c] is a dynamic technique used for predicting this characteristic. If the results of propagation analysis suggest that the *cancellation* of data state errors is likely to occur if data state errors are created, then sensitivity analysis produces results predicting a lower testability than if cancellation of data state errors were unlikely to occur.

When all of the data state errors that are created during an execution are cancelled, program failure will not occur. If this occurs repeatedly, this produces an inflated confidence that the software is *correct*. It might seem desirable for a correct output to be produced regardless of how the program arrived at the correct output. This *is* the justification for fault-tolerant software. But for critical software, any undetected fault is undesirable, even if the data state error it produces is frequently cancelled. For critical software, we prefer correct output from correct programs, not correct output from incorrect programs. By the fact the program is incorrect, there exists at least one input on which program failure will occur, and by the fact the software is critical, the potential for a loss-of-life exists.

This paper presents empirical observations concerning a phenomenon that occurs during program execution; this phenomenon suggests the likelihood of data state error cancellation

occurring. The degree to which this phenomenon occurs can be quantified by static program analysis, inspection of a specification, or both. Note that this phenomenon can be quantified statically, which is far less expensive to perform than the dynamic propagation analysis. Thus through static program analysis or specification inspection, insight is acquired concerning the likelihood that data state error cancellation will occur. And this gives insight into whether faults will remain undetected during testing, i.e., program testability.

I term this phenomenon "information loss." *Information loss* occurs when internal information computed by the program during execution is not communicated in the program's output. Information loss increases the potential for the cancellation of data state errors and this decreases software testability. As mentioned, information loss can be observed by both static program analysis and inspection of a specification. I divide information loss into two broad classes: implicit information loss and explicit information loss. Static program analysis is used to quantify the degree of explicit information loss, and specification inspection quantifies the degree of implicit information loss.

Explicit information loss occurs when variables are not validated either during execution (by a self-test) or at execution termination as output. The occurrence of explicit information loss can be observed using a technique such as static data flow analysis [Korel87]. Explicit information loss frequently occurs as a result of information hiding [Parnas72], however there are other factors that can contribute to it. Information hiding is a design philosophy that does not allow information to leave modules that could potentially be misused by other modules. Information hiding is a good design philosophy; however, it is not necessarily good for testability, because the data in the local variables is lost upon exiting a module. In Section 3.3, I propose a scheme where information hiding is kept as a part of the software design philosophy while its negative effect, explicit information loss, is lessened.

Implicit information loss occurs when two or more different incoming parameters are presented to a user-defined function or a built-in operator and produce the same outgoing parameter. An example is the integer division computation **a := a div 2**. In the computation **a := a + 1**, there is no implicit information loss. In these two examples, the potential for implicit information loss occurring is observed by statically analyzing the code. If a specification states that ten floating-point variables are input to an implementation, and 2 boolean variables contain the implementation's output, then we know that implicit information loss will occur in an implementation of this specification. Thus specifications may also hint at some degree of the implicit information loss that will occur if they are written with enough information concerning their domains and ranges.

2 Information Loss

I have proposed two broad classes of information loss. The following pseudo-code example contains both types of information loss and demonstrates how we can statically observe where these two types of information loss occur. For this example, I assume inputs **a** and **c** have effectively infinite domains, and **z** has an effectively infinite domain immediately before the statement **z := z mod 23** is executed.

```
Module x(in-parameter a : real, in-parameter c : real,
         out-parameter  b : boolean)
```

```
local-parameters
  z : integer
  y : boolean

Beginning of Body
     :
     :
z := z mod 23
     :
     :
b := f(a,c,y,z)
End of Body
```

With the assumption of effectively infinite domains for **a**, **c**, and **z**, module **x** suffers from both implicit information loss and explicit information loss. Explicit information loss occurs in **x** as a result of its 2 local variables whose values are not output nor passed out. Implicit information loss can be observed in several ways. The first way is the impossibility of taking **b**'s value at module termination and discovering the values of **a** and **c** that were originally passed in; infinitely many combinations of **a** and **c** map to a particular **b**. This potentially could have been observed from the specification of the module. The second way implicit information loss occurs is at the statement containing the **mod** operator; implicit information loss occurs because of the assumption that **z** has an effectively infinite domain. Many values of **z** map to a particular value in [0..22] after the computation.

2.1 Implicit Information Loss

Clues suggesting some degree of the implicit information loss that may occur during execution may be visible from the program's specification; I use a specification metric termed the "domain/range ratio" for suggesting a degree of implicit information loss [Voas91a]. Recall that in the example we were also able to observe implicit information loss by code inspection. Therefore, a specification's domain/range ratio only suggests a portion of the implicit information loss that may occur; code inspection can give additional information concerning implicit information loss.

The *domain/range ratio* (DRR) of a specification is the ratio between the cardinality of the domain of the specification to the cardinality of the range of the specification. I denote a DRR by $\alpha : \beta$, where α is the cardinality of the domain, and β is the cardinality of the range. As previously stated, this ratio will not always be visible from a specification. After all, there are specifications whose ranges are not known until programs are written to implement the specifications. And if the program is incorrect, an incorrect DRR will probably be calculated.

DRRs roughly predict a degree of implicit information loss. Generally as the DRR increases for a specification, the potential for implicit information loss occurring within the implementation increases. When α is greater than β, previous research has suggested that faults are more likely to remain undetected (if any exist) during testing than when $\alpha = \beta$ [Voas91a].

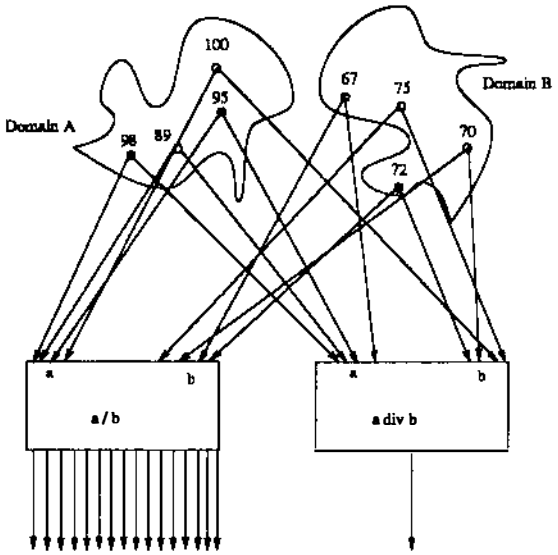


Figure 1: There are four potential values for variable a and four potential values for variable b, for a total of 16 pairs of potential inputs. Notice that for these 16 inputs, integer division always produces the same output (1), and real division produces 16 unique outputs.

The granularity of a specification (or functional description) for which we can determine a DRR varies. For example, DRRs exist for unary operators, binary operators, complex expressions, subspecifications, or specifications. (By *subspecification*, I mean a specification for what will become a *module*.) In the example, the DRR of subspecification x is $(\infty_R)^2 : 2$ and $\infty_I : 23$ for the **mod** operator. In this paper, the symbol ∞_I denotes the cardinality of the integers, and ∞_R denotes the cardinality of the reals.

For certain specifications, the inputs can be found from the outputs by inverting the specification. For example, for an infinite domain, the specification $f(x) = 2x$ has only one possible input x for any output $f(x)$. Other specifications, for example $f(x) = \tan(x)$, can have many different x values that result in an identical $f(x)$; i.e., $\tan^{-1}(x)$ is not a one-to-one function. All inverted specifications that do not produce exactly one element of the domain for each element of the range lose information that uniquely identifies the input given an output. Restated, many-to-one specifications mandate a loss of information; one-to-one specifications do not. This is another way of viewing implicit information loss.

When implicit information loss occurs, you run a risk that the lost information may have included evidence of incorrect data states. Since such evidence is not visible in the output, the probability of observing a failure during testing is somewhat reduced. The degree to which it is reduced depends on whether the incorrect information is isolated to bits in the data state that are not lost and are eventually released as output. As the probability of observing a failure decreases, the probability of undetected faults existing increases.

Another researcher who has apparently come to a similar conclusion concerning the relationship between faults remaining undetected and the type of function containing the fault is Marick [Marick90]. While performing mutation testing experiments with boolean functions, Marick [Marick90] noted that faults in boolean functions (where the cardinality of the range is of course 2) were more apt to be undetected. Boolean functions have a great degree of

	Function	Implicit Information Loss	DRR	Comment
1	$f(a) = \begin{cases} 0 & \text{if } a < 0 \\ a & \text{otherwise} \end{cases}$	yes	$\infty_I : \infty_I/2$	a is integer
2	$f(a) = a + 1$	no	$\infty_I : \infty_I$	a is integer
3	$f(a) = a \bmod b$	yes	$\infty_I : b$	testability decreases as b decreases
4	$f(a) = a \text{ div } b$	yes	$\infty_I : \infty_I/b$	testability decreases as b increases, $b \neq 0$
5	$f(a) = \text{trunc}(a)$	yes	$\infty_R : \infty_I$	a is real
6	$f(a) = \text{round}(a)$	yes	$\infty_R : \infty_I$	a is real
7	$f(a) = \text{sqr}(a)$	no	$2 \cdot \infty_R : \infty_R$	a is real
8	$f(a) = \text{sqrt}(a)$	no	$\infty_R : \infty_R/2$	a is real
9	$f(a) = a/b$	no	$\infty_R : \infty_R$	a is real, $b \neq 0$
10	$f(a) = a - 1$	no	$\infty_I : \infty_I$	a is integer
11	$f(a) = \text{even(a)}$	yes	$\infty_I : 2$	a is integer
12	$f(a) = \sin(a)$	yes	$\infty_I : 360$	a is integer (degrees), $a \geq 0$
13	$f(a) = \tan(a)$	yes	$\infty_I : 360$	a is integer (degrees), $a \geq 0$
14	$f(a) = \cos(a)$	yes	$\infty_I : 360$	a is integer (degrees), $a \geq 0$
15	$f(a) = \text{odd(a)}$	yes	$\infty_I : 2$	a is integer
16	$f(a) = \text{not(a)}$	no	$1 : 1$	a is boolean

Table 1: DRRs and implicit information loss of various functions.

implicit information loss. This result compliments the idea that testability and the DRR are correlated. Additional evidence that correlation exists between implicit information loss and testability is currently being collected.

2.1.1 Correlating Implicit Information Loss and the DRR

Implicit information loss is common in many of the built-in operators of modern programming languages. Operators such as **div**, **mod**, and **trunc** have high DRRs.

Table 1 contains a set of functions with generalized degrees of implicit information loss and DRRs. A function classified as having a *yes* for implicit information loss in Table 1 is more likely to receive an altered incoming parameter and still produce identical output as if the original incoming parameter were used; a function classified as having *no* implicit information loss in Table 1 is one that if given an altered incoming parameter would produce altered output. A *yes* in Table 1 suggests data state error cancellation would occur; a *no* suggests data state error cancellation would not occur. In Table 1, all references to b assume it be a constant for simplicity. This way we only have to deal with the domain of a single input, instead of the domain of a 2-tuple input. The infinities in the table are mathematical entities, but for any computer environment they will represent the cardinality of fixed length number representations of finite size.

Instead of describing the generalizations made concerning implicit information loss for each element of Table 1, Figure 1 illustrates the relationship between implicit information loss and the DRR. In Figure 1, we have 16 (a,b) input pairs that are presented to 2 functions: one performs real division, the other performs integer division. For the real division function there are 16 unique outputs, and for the integer division function there is one output. This example shows how the differences in the DRRs of these two forms of division are correlated to different amounts of information loss.

2.2 Explicit Information Loss

Explicit information loss is not predicted by a DRR as implicit information loss is. Recall that explicit information loss is observed through code inspection, whereas the potential for implicit information loss can be predicted from functional descriptions or code inspection. Explicit information loss may also be observable from a design document depending on its level of detail. Explicit information loss is more dependent on how the software is designed, and less dependent on the specification's (input, output) pairs.

2.2.1 Observability

Integrated circuit design engineers have a notion similar to explicit information loss that they term "observability." *Observability* is the ability to view the value of a particular node that is embedded in a circuit [Markowitz88]. When explicit information loss occurs in software, you loose the ability to see information in the local variables. So in this sense, greater amounts of explicit information loss in software is a parallel to lower observability in circuits.

Discussing the observability of integrated circuits, [Berglund79] states that the principal obstacle in testing large-scale integrated circuits is the inaccessibility of the internal signals. One method used for increasing observability in integrated circuits design is to increase the pin count of a chip, allowing the extra pins to carry out additional internal signals that can be checked during testing. These output pins increase observability by increasing the range of potential bit strings from the chip. In Section 3.3, I propose applying a similar notion to increasing the pin count during software testing—increasing the amount of data state information that is checked during testing. Another method used for increasing observability is inserting internal probes to trap internal signals; Section 3.3 also proposes a similar technique by self-testing internal computations during execution.

3 Design Heuristics

Section 3 presents several strategies for lessening the effects of information loss. Section 3.1 describes benefits gained during program validation if specifications are decomposed in a manner that lessens the effect of implicit information loss at the specification level. Section 3.2 describes a way of lessening the effect of implicit information loss at the implementation level. Section 3.3 describes ways of lessening the effects of explicit information loss.

3.1 Specification Decomposition: Isolating Implicit Information Loss

Although the DRR of a specification is *fixed* and cannot be modified without changing the specification itself, there are ways of decomposing a specification that lessen the potential of data state error cancellation occurring across modules. During specification decomposition, you have hands-on control of the DRR of each subfunction. With this, you gain an intuitive feeling (before a subfunction is implemented) for the degree of testing needed for a particular confidence that a module is propagating data state errors. The rule-of-thumb that guides this intuitive feeling is: "the greater the DRR, the more testing needed to overcome the potential for data state error cancellation occurring."

Section 3.1 presents a benefit that can be gained for testing purposes by using a specification's DRR during design. During a design, a specification is decomposed in a manner such that the program's modules are designed to either have a high DRR or a low DRR. By isolating modules that are more likely to propagate incoming data state errors through them during program testing (low DRR), testing resources can be shifted during module testing to modules that are less likely to propagate incoming data state errors across them.

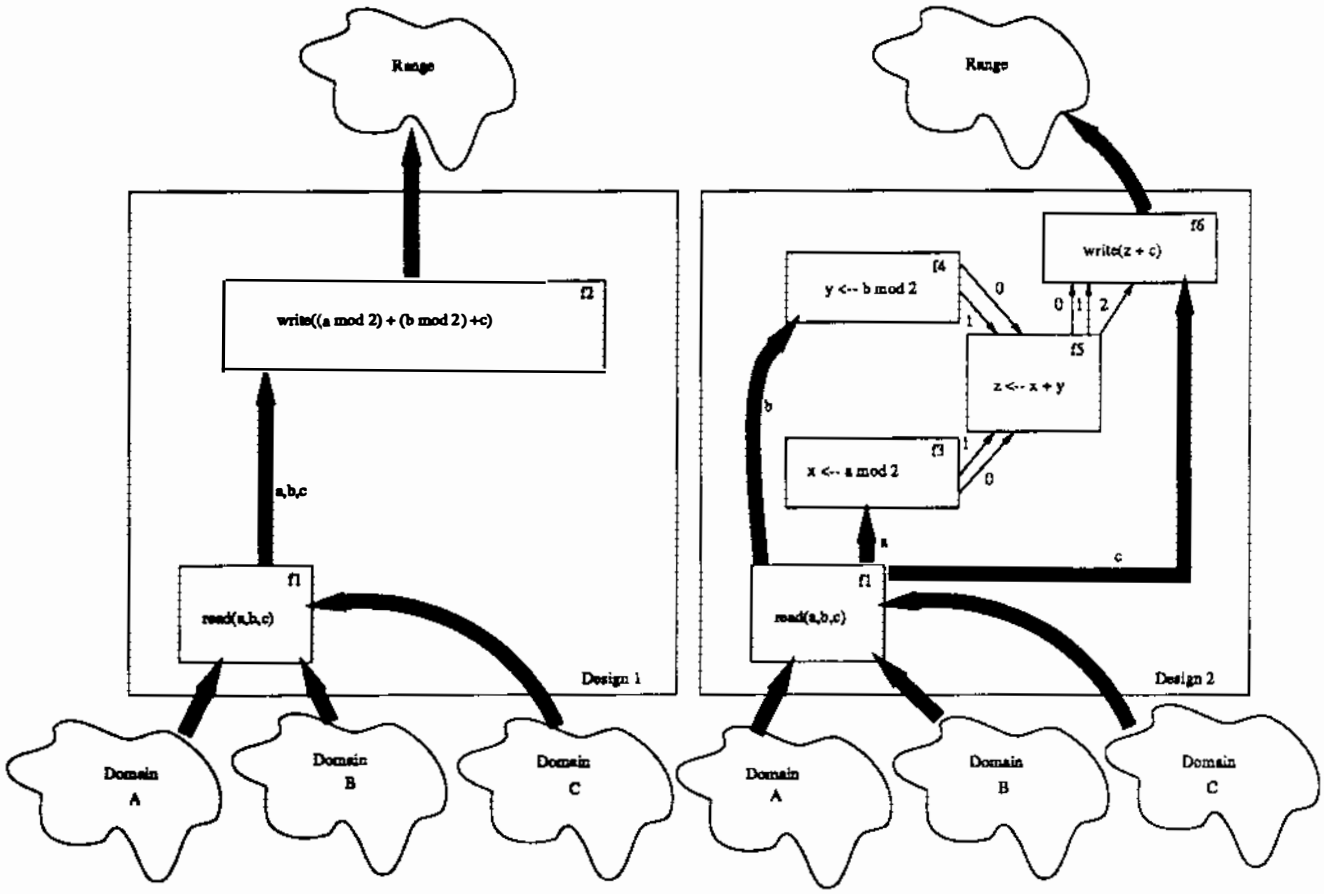
I am not suggesting that specification decomposition in this manner is always possible, but rather when possible, it can benefit those persons testing the program. By isolating higher amounts of implicit information loss, the benefit derived is knowing which sections of a program have a greater ability to cancel incoming data state errors before testing begins. This provides insight for where testing is more critically needed. This allows testers to shift testing resources from sections needing less effort to sections needing more.

As an example, consider a specification g:

$$g(a,b,c) = \begin{cases} c+2 & \text{if odd}(a) \text{ and odd}(b) \\ c+1 & \text{if odd}(a) \text{ or odd}(b) \\ c & \text{otherwise} \end{cases}$$

where a, b, and c are integers. Many different designs can be used to compute g, but I will concentrate on two designs are also shown in the table in Figure 2: Design 1 and Design 2 (In Figure 2 a thick arc represents large sets of values (too many to enumerate), and a thin arc represents a single value.) The DRR of g is $\infty_I^3 : \infty_I$. The DRRs of the subfunctions of Designs 1 and 2 are shown in Figure 2. Design 1 has two subfunctions, $f1$ and $f2$. In Design 2, I have taken g with its DRR of $\infty_I^3 : \infty_I$ and have decomposed it in such a manner as to isolate the subfunctions that create its high DRR: $f3$ and $f4$. This decomposition provides *a priori* information concerning where to concentrate testing (in $f3$ and $f4$) and where not to (in $f5$, since subfunction $f5$ can be exhaustively tested). Had subfunction $f5$ not been separated out, then in whatever other design this computation occurred, it would be needlessly retested.

The reader might ask why subfunctions $f3$ and $f4$ should receive additional testing. This is because if anything were to occur to the values of variables **a** and **b** before subfunctions $f3$ and $f4$ are executed (thus causing a data state error affecting these variables), it is likely that these subfunctions will cancel the data state error. We should test less in $f5$ and test more in $f3$ and $f4$. This shows how isolation according to module DRRs can benefit testing.



subfunction	classification	DRR
$f1$	VDVR	$\infty_I^3 : \infty_I^3$
$f2$	VDVR	$\infty_I^3 : \infty_I$
$f3$	VDFR	$\infty_I : 2$
$f4$	VDFR	$\infty_I : 2$
$f5$	FDFR	4:3
$f6$	VDVR	$3 \cdot \infty_I : \infty_I$

Figure 2: Design 1 (left); Design 2 (right).

3.2 Minimizing Variable Reuse: Lessening Implicit Information Loss

A method for decreasing the amount of implicit information loss that occurs at the operator level of granularity is *minimizing* the reuse of the variables. For instance, as we have already seen, a computation such as **a := sqr(a)** destroys the original value of **a**, and although you can take the square root after this computation and retrieve the absolute value that a had, you have lost the sign. Minimizing variable reuse is one attempt to decrease the amount of implicit information loss that is caused by built-in operators such as **sqr**.

Minimizing variable reuse requires either creating more complex expressions or declaring more variables. If the number of variables is increased, memory requirements are also increased during execution. If complex expressions are used, we lessen the testability because a single value represents what were previously many intermediate values. Although there is literature supporting programming languages based on few or no variables [Backus78], programs written in such languages will almost certainly suffer from low testabilities. Thus I advocate declaring more variables.

3.3 Increasing Out-Parameters: Lessening Explicit Information Loss

Consider the analogy where modules are integrated circuits and local variables are internal signals in integrated circuits. This analogy allows us to see how explicit information loss caused by local variables parallels the notion of low observability in integrated circuits. Since explicit information loss suggests lower testabilities, I prefer, when possible, to lessen the amount of explicit information loss that occurs during testing. And if limiting the amount of explicit information loss is not possible, I at least have the benefit of knowing where the modules with greater data state error cancellation potential are before validation begins.

One approach to limiting the amount of explicit information loss is to insert **write** statements to print internal information. This information must then be checked against the correct information. A second approach is increasing the amount of output that these subspecifications return by treating local variables as out-parameters. A third approach inserts self-tests (this is similar to the assertions suggested in [Shimeall91] for fault detection) that are executed to check internal information during computation. In this approach, messages concerning incorrect internal computations are subsequently produced.

These approaches produce the same end results, however in the processes employed to achieve these results they differ slightly. The end results of these approaches are:

1. Forcing those persons involved in the formalization of a specification to produce detailed information about the states of the internal computations. This should increase the likelihood that the code is written correctly.

2. Increasing the cardinality of the range.

As an example of the third approach, consider inserting self-tests into the declaration given in Section 2:

```
Module x(in-parameter a : real, in-parameter c : real,
         out-parameter  b : boolean)

local-parameters
  z : integer
  y : boolean

Beginning of Body
    :
    :
z := z mod 23
self-test(z,ok)
if not(ok) then write('warning on z')
    :
    :
    :
y := expression
self-test(y,ok)
if not(ok) then write('warning on y')
    :
b := f(a,c,y,z)
End of Body
```

A self-test such as **self-test(z,ok)** may either state explicitly what value z should have for a given (a,c) pair, or it may give a range of tolerable values for **z** in terms of a particular (a,c) pairing. If a self-test fails, a warning is produced.

These three approaches simulate the idea previously mentioned that is used in integrated circuits—increasing the observability of internal signals [Berglund79, Markowitz88]. In these approaches, I am not discrediting the practice of information hiding during design. However, when writing software such as safety-critical software, there is a competing imperative: to enhance testability. Information that is not released encourages undetected faults, and increased output discourages undetected faults.

The downside to these approaches is that for the approaches to be beneficial, they all need additional specified information concerning the internal computations. Maybe the real message of this research is that until we make the effort to better specify what must occur, even at the intermediate computation level, testabilities will remain lower.

3.4 Combining Approaches

We have seen how different techniques can be used against various classifications of information loss. An even better methodology for achieving this goal is a combination of techniques, applied at both design and implementation phases. For example, combining the technique of releasing more internal information with the technique of minimizing variable reuse furthers the available information for validation. The limit to any combined approach, however, will be the ability to validate the additional information. After all, if the additional information can not be validated, then there is no reason to expose it.

4 Summary

Information loss is a phenomenon to be considered by those who gain confidence in the correctness of software through software testing. The suggestion that information loss and testability are related is important; it implies that the ability to gain confidence in the absence of faults from observing no failures may be limited for programs that implement functions that encourage information loss. Although discouraging on the surface, I feel that there are ways to lessen this limitation with prudent design and implementation techniques.

The unfortunate conclusion of Section 3 is that we must validate more internal information if we hope to increase software testability. To validate more internal information, we must have some way of checking this additional internal information. This requires that more information be specified in the specification or requirements phase. And for certain applications this information is rarely available.

It may be that a theoretical upper bound exists on the testability that can be achieved for a given (functional description, input distribution) pair. If we can change the functional description to include more internal information, we should be able to push the upper bound higher. Although the existence of an upper bound on testability is mentioned solely as conjecture, my research using sensitivity analysis and studying software's tendency to not reveal faults during testing suggests that such exists. I challenge software testing researchers to consider this conjecture.

5 Acknowledgement

This research has been supported by National Research Council NASA-Langley Resident Research Associateship.

References

[Backus78] J. Backus. Can Programming Be Liberated from the Von Neumann Style? A Functional Style and its Algebra Programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Berglund79] Neil C. Berglund. Level-Sensitive Scan Design Tests Chips, Boards, System. *Electronics*, March 15 1979.

[Korel87] Bodgan Korel. The Program Dependence Graph in Static Program Testing. *Information Processing Letters*, January 1987.

[Marick90] Brian Marick. Two Experiments in Software Testing. Technical Report UIUCDCS-R-90-1644, University of Illinois at Urbana-Champaign, Department of Computer Science, November 1990.

[Markowitz88] Michael C. Markowitz. High-Density ICs Need Design-For-Test Methods. *EDN*, 33(24), November 24 1988.

[Parnas72] David L. Parnas. On Criteria to be used in Decomposing Systems into Modules. *Communications of the ACM*, 14(1):221–227, April 1972.

[Shimeall91] Timothy J. Shimeall and Nancy G. Leveson. An Empirical Comparison of Software Fault Tolerance and Fault Elimination. *IEEE Transactions on Software Engineering*, 17(2):173–182, February 1991.

[Voas91a] J. Voas and K. Miller. Improving Software Reliability by Estimating the Fault Hiding Ability of a Program Before it is Written. In *Proceedings of the 9th Software Reliability Symposium*, Colorado Springs, CO, May 1991. Denver Section of the IEEE Reliability Society.

[Voas91b] J. Voas, L. Morell, and K. Miller. Predicting Where Faults Can Hide From Testing. *IEEE Software*, 8(2), March 1991.

[Voas91c] J. Voas. A Dynamic Failure Model for Estimating the Impact that a Program Location has on the Program. In *Proceedings of the 3rd European Software Engineering Conf.*, Milano, Italy, October 1991.

MOTHER: A Test Harness for a Project with Volatile Requirements

Joe Maybee

Graphic Printing and Imaging Division
Tektronix, Incorporated
Mail Stop 63-424
P.O. Box 1000 Wilsonville, Or. 97070
Usenet: maybee@pogo.WV.TEK.COM
Phone: 685-3572

ABSTRACT

This paper describes experiences with a test harness designed to quickly accommodate changing external requirements. This paper is intended to share experiences and ideas for automated testing with quality assurance engineers. The system under test is a real-time embedded system that controls the complex electromechanical mechanisms in a graphic printer.

Keywords and Phrases: real-time embedded systems, automated test methods, requirements testing, electromechanical systems.

Biographical: Joe Maybee is a Senior Software Engineer with the Graphic Printing and Imaging Division at Tektronix. Joe has been employed as an engineer with Tektronix since 1978, and has specialized in the design and implementation of real-time embedded systems during this time. Recently, Joe has focused on software quality in real-time embedded systems and the accurate definition of user requirements.

CONTENTS

MOTHER: A Test Harness for a Project with Volatile Requirements

Joe Maybee

Graphic Printing and Imaging Division
Tektronix, Incorporated
Mail Stop 63-424
P.O. Box 1000 Wilsonville, Or. 97070
Usenet: maybee@pogo.WV.TEK.COM
Phone: 685-3572

1. Introduction

1.1 The basic nature of the problem to be solved: changing requirements

Software engineers have long based procedures and methods upon the existence of concise, immutable requirements. The truth of the matter is that requirements are seldom, if ever, the static, universal truths that we wish them to be.

In the classic software models, such as the *waterfall* model of software engineering, changing requirements cause a ripple effect that cause perturbations throughout the engineering process ''pipeline''.

Regardless of how we wish that requirements were unchanging, the brutal truth is that freezing a requirements specification to achieve a smooth, textbook software engineering process can result in an engineering process that produces a high quality product that no one wants to buy. The fact that requirements are volatile is a reality built into the environment in which today's products must compete: customers' requirements are changing at a faster rate than ever before.

This paper describes how one *Quality Assurance* (QA) group attacked the problem of changing requirements.

1.2 The nature of the problem from the QA viewpoint: changing tests

From the QA viewpoint, the problem becomes one of reacting to the changing requirements on demand. Providing facilities for quickly modifying, adding, or deleting the defined set of requirements *and* their associated tests allows the QA team to respond to changing requirements in a timely fashion.

Reacting quickly to change becomes a relatively easy task, provided that both of the following are true:

1. The QA team has control over the source documents for the specifications.
2. The requirements and tests are tightly coupled.

1.3 The tight coupling of requirements and tests

The concept of tight coupling of requirements and tests is not new. Many other QA teams have explored the advantages of having requirements where tests are easily traceable to their associated requirements and vice versa.

Traceability of the tests and requirements is only a part of the concept of tight coupling, however. To take advantage of traceability, it becomes necessary for the test writer to be able to quickly find the requirement for a given test. In the case where a requirement is being changed or deleted, it becomes necessary for the requirement writer to quickly locate and then change or delete the associated test.

1.4 Giving implementation engineers the timely response they need

It is very difficult to schedule discoveries. Since discovery is an integral part of diagnosing the source of bugs, it follows that it is difficult to schedule bug fixes. In many instances, the discovery of the bug is only a part of the problem: creative solutions are often needed to fix the bug.

From a QA standpoint, scheduling QA runs is less of a problem because they tend to be quantifiable tasks.

When scheduling projects, the verification portion of the project is almost always scheduled as a fixed-length task, and it is always the most stressful: magazine advertisements are bought, conferences are planned, and manufacturing is gearing up for production runs. Software schedule slips at the end of projects are the *most* reprehensible, since they disrupt *everyone's* schedule.

Given that the QA task is the easiest to schedule and manage, it becomes most prudent to make the QA task as fast and as efficient as possible. Every minute saved in the QA process is a minute that can be given to the design engineering team for bug isolation and fixes.

1.5 Changing the test harness for the test writers

In many instances, new requirements for the product can place new requirements on the test harness. It is necessary, therefore, to provide the test writers with the ability to define new test operations in a rather cavalier manner.[1]

Making the test harness as flexible as possible to accommodate the test writers is, therefore, a derived requirement.

[1] I once read an album cover for a Jazz album that referred to an improvisation as the TMIUATGA technique: "They're Making It Up As They Go Along." This concept seems to fit here rather nicely.

2. The design of a quick-turnaround test system

Designing a quick-turnaround test harness is not as difficult as one might expect: our test harness was assembled and made functional in a matter of weeks by two engineers.

Such a harness need not be expensive, either. Our test harness was assembled using some inexpensive, simple, off-the-shelf tools.

We managed to assemble this cheap, effective test harness by looking at the problem *carefully* and from a slightly different perspective than usual.

2.1 The problem of changing requirements

The problem of fast response to changing requirements becomes a problem in fast manipulation of source documents for the *Software Requirements Specification* (SRS). The SRS needs to be easily accessible and easily modified by the persons responsible for the upkeep of this document.

Under ideal conditions, the SRS is a repository of all external requirements for the system software.

2.1.1 Tight coupling of requirements and tests

Some of problems with changing requirements are that:

1. The changed requirement is passed to team members in an informal fashion, and is never recorded in the SRS. This usually happens in later stages of the project when the SRS is perceived as having outlived its usefulness.
2. The test for the changed requirement is never updated. In the case of a dropped requirement, sometimes the test is not removed from the test suites. In the case of an added requirement, sometimes the appropriate test is never added.

Any suitable test harness design will provide for the inherent tight coupling between the written requirements and the test for that requirement.

2.1.1.1 Making requirements writing a QA responsibility

Traditionally, writing the SRS has been a task that belonged to the design engineering team. After the SRS was written, the QA team was then faced with the problem of isolating the design requirements from this document. Often, these requirements were less than explicit.

It is difficult to find fault with the design engineers for the problems that arise with the SRS, since in many instances the true purpose of this document is less than clear.

Testability is a measure of a requirement's quality. If a requirement isn't testable, there is no point in defining it as a requirement.

The QA group has a vested interest in the SRS, and as such, should be more than happy to take charge of such a document. Design engineering, on the other hand, should be more than happy to relieve itself of this responsibility.[2]

[2] In the early stages of this project, I had engaged a member of the design team in a heated debate on specifications. I had generated a list of documents that were to be written and their specific audiences and purpose. I was told by the designer that if I had such a definite idea as to how the SRS should be written, why didn't I assume the burden of writing it? Of course, I seized the opportunity.

2.1.1.2 Embedding the tests in the requirements document

A simple method for binding the tests to the requirement is to embed the test in the SRS immediately after the requirement. As long as the two can be correctly identified within the document, this provides a simple coupling method that provides the necessary link for finding either the test or the requirement. If you want to find the test for the requirement, look immediately after the requirement. If you want to find the requirement for a particular test, look in front of the test.

By breaking the requirements document up into logical, manageable pieces, it is possible to provide the granularity necessary in order to allow several QA engineers to work on different parts of the document at once.

2.1.1.3 Making it easy to write tests

By using a test description "language"[3] of our own design, we can provide the QA engineers with a means for writing abstract descriptions of tests. If our test description "language" is carefully designed, it will be easy for QA engineers to remember the necessary "vocabulary."

In instances where our test "vocabulary" is inadequate, the test harness should allow for the fast installation of extensions. In short, it ought to allow QA engineers to "make it up as they go along" in instances where the current test "vocabulary" fails them.

2.1.1.4 Enforcing the disciplines

To enforce the disciplines required of such a scheme, here we would simply advise: use QA engineers, not design engineers. QA engineers will most likely have more of a vested interest in maintaining the requirements and tests than the design engineers will. There seems to be a tendency among design engineers to abandon documents almost immediately after they are written, since document maintenance seems like "living in the past."[4]

Enforcing discipline becomes an oxymoron, of sorts. Discipline either exists or it doesn't, and cannot be "enforced."

We have made it as easy as possible to find both the test *and* the requirement: they are always together. If you change one, you change the other. If QA engineers are not inclined to practice good document maintenance procedures, all is probably lost anyway.

2.1.2 Timely turnaround of the QA testing procedure

There are several necessary items for timely turnaround in the QA procedure:

1. Fast evaluation and correction of tests: are they written correctly?
2. Fast extraction of the latest tests: getting the embedded tests out of the current version of the SRS as quickly as possible.
3. Fast execution of the latest tests: getting the tests executed as quickly as possible.
4. Fast evaluation of the test results: getting the test exceptions report generated as quickly as possible.

[3] The language at use here is the *FORTH* language with a set of custom operators. As the reader will see later on it provides an almost prose-like "language" for writing descriptions of tests for the harness.

[4] Before any design engineers take out any contracts, I'd like to point out that I, myself, am a design engineer. I understand how this works, since I have been in this position many times. Remember, I'm on *your* side: did you read the part where I said the QA team should write the requirements specification?

2.1.2.1 Fast evaluation and correction of tests

QA engineers are humans.[5] Humans make mistakes in interesting ways. It becomes necessary to be able to quickly evaluate the validity of a test in terms of correct use of the test "language" and "vocabulary." If the test harness is sufficiently fast, it is trivial to check the tests: simply feed the test to the test harness and look at the results.

Adopting this approach places additional emphasis on the need to make the test harness operate as quickly as possible.

2.1.2.2 Fast extraction of the latest tests

We have indicated in prior discussion that the design process becomes more difficult to schedule near the end of the project. The same is true for the design of the test cases. After we check a particular run of the latest tests for test errors, it becomes desirable to generate the next test suite from the corrected SRS as soon as possible.

If we place sufficient emphasis on the efficiency of the test extraction software, we can generate the new test suites in a modest amount of time. This turns out to be reasonably simple to do, provided we choose our text formatter and SRS test "vocabulary" wisely.

By providing keyword delimiters that identify the beginning and end of a requirement and the beginning and end of a test, the test extraction software merely has to scan the file and generate the necessary files based on these delimiters.

2.1.2.3 Fast execution of the latest tests

Fast execution of the latest test suite becomes necessary for quick response to new or changed tests *and* to new versions of the product software. In this case, direct execution of the test cases by an interpreter is, in fact, extremely practical.

We adapted a public domain FORTH interpreter to our purpose.[6] Since the FORTH interpreter was written in C, we were able to install our own operators in the interpreter to drive a set of off-the-shelf boards from an instrumentation company. With the assistance of an electrical engineer, we were able to interface these boards with the printer under test.

By using a *state-stimulus-response* approach to our requirements, we are able to write tests with reasonable ease. Using an approach where the requirements were categorized within the SRS along firmware inputs, we were able to specify the response of the printer to every stimulus across all operational modes. While this is an exhaustive approach, it has the advantage of providing excellent coverage in the specification and has the added benefit of accentuating incomplete areas: the SRS is not finished until a response is defined for *every* stimulus across *all* operational modes.

---

[5] There is obvious room for debate here. A QA team I know of was once accused of using an "army of monkeys" testing strategy by a second-level manager whose own heritage had been called into question more than once. A picture clipped from a newspaper of a sheep with its hoof on a terminal keyboard appeared in the QA area. A hand-written caption under the sheep read: "We will no longer use an army of monkeys to test software."

[6] I snagged this FORTH interpreter off of a *USENET* news group (comp.misc.sources), and had it laying around in a directory when this opportunity presented itself. If you're trying to justify a *USENET* connection to your manager, point out that opportunity favors the prepared pack-rat.

By using the *state-stimulus-response* approach to specification, our initial guess at the necessary operators was that there would have to be five major categories of operators:

1. Operators to put the printer into a particular state.
2. Operators to provide the printer with a particular stimulus.
3. Operators that monitor the printer for a particular response.
4. Operators that provide elementary utilities, such as a "deadman timer" (sometimes called a "watchdog timer") to keep response operators from waiting forever for a response from a "dead" printer.
5. TMIUATGA[7] Operators: Operators that we never even dreamed of.

Our first guess proved reasonably fruitful: it provided over half of the necessary operators.

Also, by providing logging facilities built into the modified interpreter it became very easy to log the progress of the test. Log files were generated for:

1. The requirements themselves. Requirements were embedded in the test suites and are displayed upon the screen of the test harness console as the test is being run.
2. The results of the tests. Every operator in the test suite left a value on the FORTH stack that indicated the success, failure, or timeout of that operation. At the end of the test, if there was anything other than successes on the stack, the test was marked in this log as FAILED (along with the requirement number that failed).
3. Any input from the diagnostic port of the printer under test. Design engineers sometimes wrote important diagnostic information out an RS-232 port on the machine, which we were more than happy to capture and log for them.
4. Any operator instructions that were issued. In the course of testing a printer, it becomes necessary to inspect the resulting prints. In some cases, the test needs to instruct the test harness operator to write an identifying number on the print and lay it aside for inspection by the test analyst. In other cases, the test harness needs to tell the test harness operator to remove jammed or misfed paper from the machine.
5. The test stream itself.

Each test log contains a sequence number that is incremented *anytime* a message is written to *any* file, a time and date stamp, the identifier of the requirement currently under test and the message itself. This makes it possible to extract, merge and sort any combination of the logs to produce the reports discussed in the next section.

2.1.2.4 Fast evaluation of the test results

Fast evaluation of test results is also a critical element of the fast turnaround requirement for the test harness.

A simple utility is written to search the logs for errors, which have been carefully tagged by the test harness with the words "FAILED" or "TIMED OUT". Since these records contain requirement numbers, records containing information germane to this failure can be extracted from the other logs by requirement number, then sorted by the leading sequence number to produce accurate sequential audit trails as to what occurred during the test itself.

[7] "They're Making It Up As They Go Along" .... Remember?

2.2 The problem of changing tests

Several problems are posed by changing requirements or tests. Most of these are minor problems, by design. The problem of changing tests is met by the following derived requirements:

1. The test harness needs to be extensible.
2. The tests will have to be evaluated and corrected.

2.2.1 The test harness needs to be extensible

The test harness is indeed extensible. The addition of any new operator simply requires the addition of a small amount of C code, or definition of the new operator using a combination of existing operators in FORTH.

2.2.1.1 New test operators will be required as needs are discovered

There will be new test operators required as the test writers discover new operations that will need to be performed. The approach we used was simple: make up the operators that are needed, then they can be installed by a test technician with a minimum amount of effort.

2.2.1.2 New test operators will have to be tested

The new operators can be tested in the test harness itself. The test harness is designed to take FORTH input from the keyboard as well as from file input. The QA engineer testing the new operator can key in an experimental sequence from the console to generate a test case for the new operator.

2.2.2 New tests will have to be evaluated and corrected

As mentioned previously, because of the quick turnaround from the test harness itself, the best way to evaluate new tests is to simply *run them.*

Fast evaluation of the results provide the QA engineer with the feedback necessary to evaluate the tests in a modest amount of time, usually a matter of an hour or two. Once a suite is corrected, it only needs to be retested if more modifications are made.

2.2.2.1 Test operators will be misspelled

There are two approaches to this problem:

1. Correct the misspelling in the test suite.
2. Add another operator with the misspelling, should the number of misspellings prove too great.

Misspellings usually indicate that an operator name was poorly chosen, usually because of inconsistencies with other operator names. Many times it is better to change the name of the operator than to change the test suite: the misspelling will reappear again and again.

2.2.2.2 Test operators will be misused

Test operators will sometimes be misused. Usually this is because of a miscommunication between the test writer and the test harness implementor. Fortunately, this has happened very rarely on this project. The same approach as the previous section applies here:

1. Correct the misuse in the test suite.
2. Revise the operator to behave according to the way it tends to be used.

In almost every case, it is better to revise the operator to behave the way it tends to be used: usually the test writer has been hopelessly indoctrinated in the misuse of the operator. It may well be that the so-called "misuse" indicates a conceptual problem with the test harness implementor. Never put the cart before the horse: support the test writer, not the test harness implementor. In this case the test writer is the customer: *the customer always comes first.*

3. What is MOTHER?

3.1 Meaning of the Acronym

MOTHER is an acronym for "Maybee's Own Test Harness for Evolving Requirements." This name itself evolved from a *pet name* that the test harness earned in its early implementation stages. It took a considerable effort to come up with an name that justified the acronym.

3.2 MOTHER is based on a system composed of tools

MOTHER is hardly a monolithic system. MOTHER consists of a series of simple, but highly-specialized tools strung together with a set of scripts. These tools consist of:

1. Document generation tools.
2. Test suite generation tools.
3. Automated test tools (the MOTHER test harness).
4. Automated test exception report tools.

If the reader is more interested in the actual use of MOTHER rather than the building-blocks, I recommend skipping ahead to the section entitled *Using MOTHER*.

3.3 Document generation tools

3.3.1 The "ms" document formatter

The *ms* document formatter is a macro package for the *troff* document formatter. The *ms* package has macros that support various document formats. (This paper was generated using the *ms* macro package.)

3.3.2 "ms" based macros

The *troff* document formatter provides a macro facility that allows users to define their own macros. Since we place requirements and tests in the same file, we may define macros that delimit each requirement and its associated tests. A completely delimited requirement and test would look like this:

```
.RQ
The ready mode shall cause the FAULT line to be asserted
at the interface when the jam access door is opened.
.RE
.TS
SET-READY-MODE
OPEN JAM-ACCESS-DOOR
?FAULTED TRUE-IS-SUCCESS
.TE
```

The macro `.RQ` delimits the beginning of a requirement, while them macro `.RE` delimits the end. The `.TS` macro delimits the start of a test and the `.TE` macro delimits the end of the test.

These macros are defined with a built-in switch that allows QA engineers to print the requirements specification *with* or *without* tests included. If copies of the SRS are needed for reviewing the tests, the "*include tests*" switch can be set to cause copies of the SRS to be printed with both the requirements and their tests. If, on the other hand, the tests are not required, tremendous

amounts of paper can be saved by turning off the "*include tests*" switch.[8]

3.3.3 Source control (RCS)

Off-the-shelf source control tools were used to control the SRS. We chose the RCS package for no other reasons than: it was already there, we were familiar with it, and it did everything we needed it to do.

Primarily, we needed a facility that would allow us to:

1. Control the ownership of the files to prevent two people from working on the same file at the same time.
2. Merge the changes after two people work on different revisions of the same document at the same time.
3. Annotate revisions of the files with commentary describing the changes made to the document.
4. Review the commentary on the changes made to the various revisions of the document.
5. Review what was *really* changed in the various revisions of the document.
6. Remove *improvements* to the files that proved to be detrimental for various reasons.

3.4 Test suite generation tools

3.4.1 Smoke and mirrors: Shell and PERL scripts

Using the fundamental off-the-shelf building blocks described in the previous sections, a little "glue" is needed to piece the entire system together. Specifically, we need to be able to:

1. Extract the tests from the SRS source files.
2. Isolate tests by category: those with outstanding bugs, those that require human intervention to run, those that can run in an unattended fashion, and those that are only partially written (or not written at all).
3. Generate metrics for the test suites: How many tests fall into each of the previous categories.
4. Monitor the progress of the test harness machinery.
5. Generate the test exception reports from the test logs.

3.4.1.1 Bourne shell is the lowest common denominator

We used the *Bourne Shell* as our "glue" for our system. The reasons for this are elementary, and are part of our ongoing theme: it was already there, we were familiar with it, and it did everything we needed it to do.[9]

---

[8] The size of the SRS as it exists *without* the tests is downright intimidating. It is a reference book, not literature. Imagine trying to read a large volume of mathematical tables as if it were prose. Anything to reduce its size is desirable, especially when giving a copy to an outside organization *for review*.

[9] The reader is probably more than familiar with the advantages of this philosophy. In many instances the time required to study and analyze new tools, the time required for learning new tools, and the associated expense of new tools is prohibitive. Remember, we are striving for *fast* responses. Loosely coupled tools are highly configurable and allow the engineer access to the inner works: if a tool doesn't do *exactly* what you need to do, rewire it!

The *Bourne shell* also has the advantage of high proliferation. It's everywhere. This gives a degree of portability, but this is a moot point: portability was not one our aims.

3.4.1.2 PERL: Practical Extraction and Report Language

PERL is a report language that is gaining popularity as a replacement tool for "*sed*" and "*awk.*" PERL has two advantages as a general purpose tool for manipulating files and generating reports: it's easy to learn, and it has a very wide variety of capabilities.[10]

3.4.2 Shell script: build_test_suite

The shell script `build_test_suite` generates the entire test suite from the SRS and sorts the test files into directories on the basis of the status of the test. The status of the test may be as follows:

1. The test may be a fully automatic test. Such test files are sorted into a directory called `auto`.
2. The test may require intervention from a human operator, such as marking a test print with a test number for later inspection. Such test files are sorted into a directory called `manual`.
3. The test may be partially written. That is, a portion of the test may be waiting for the implementation of an operator that was recently "*invented.*" Such test files are sorted into a directory called `partial`.
4. The test may not be written at all. The test suite generator is set up to recognize that a requirement may not have an associated test. In this case, these test files (which consists of the requirement portion and nothing else) are sorted into a directory called `no_test`.

The first step in accomplishing these sorts is to combine the multitude of individual files in the SRS into a monolithic unit. This is accomplished by using a utility called `soelim` that comes with the *troff* package. Since the entire SRS is generated by *troff*, individual chapters are "included" into the body of the main file using the include macro " `.so`", provided by the *troff* package. The `soelim` utility expands all included files into a single output stream that is redirected to a file, thus achieving a file combining all of the latest individual files of the SRS.

The next step involves extracting the tests from the SRS. This is achieved using a PERL script called `extract_tests`. This script takes the monolithic SRS file as input, and watches for the special macro names `.RQ` (beginning of requirement), `.RE` (end of requirement), `.TS` (beginning of test) and `.TE` (end of test).

Since the `.RQ` macro generates a unique requirement number that gets printed with the requirement header, the `extract_tests` script generates the exact same number that is then used as a file name for the tests. In other words, when the document is printed, the requirement numbered 2.4.6.7.8.2 can be found with its associated test in a file *named* 2.4.6.7.8.2, and will be sorted into its appropriate directory in the next step. In the process of extracting the tests, the delimiter macros `.RQ`, `.RE`, `.TS`, and `.TE` are stripped of their leading periods, thus converting them into specialized FORTH operators that will have significance to the harness described in the next section, *Automated testing tool (MOTHER).*

[10] For an excellent series of articles on PERL, see the *Daemons and Dragons* section of *Unix Review*, Vol. 8, Nos. 5, 6 & 7. Rob Kolstad, the author of these articles, provides a very instructive tour of PERL.

The identification of test files is a simple process:

1. If the file contains a FORTH comment beginning in column one, it is presumed that this comment is substituting for an operation that is yet to be decided on. (This is a general convention used by all test writers in our group.) These tests are then sorted into the directory `partial`.
2. If the test contains a key operator that requires human intervention to complete, the test file is placed in the `manual` directory. A file of these key operators is kept in a predefined file, and is read by the `extract_tests` script.
3. If the file is missing a `TS` operator, it has no written test, and will be sorted into the directory `no_test`.
4. If the file does not fit into any of the previous categories, it is then presumed to be automatic and will be sorted into the directory `auto`.

Once all of the tests have been generated, the utility `sort_tests` sorts the tests into their appropriate directories. This proves to be useful from a document management perspective. For instance, once we find a requirement, say, 2.4.6.7.8.2 in the directory structure, we know its status. If we find 2.4.6.7.8.2 in the `no_test` directory, we know that there is currently no test associated with this requirement. This proves to be a most useful scheme when building the tests themselves, since a simple listing of the `no_test` directory gives us a listing of files we need to write tests for, and a listing of the `partial` directory gives us information about functionality still needed in the test harness.

In the next step, the `build_test_suite` calls a shell script `tag_deferred_tests` to move any tests with outstanding bugs into a deferred state. These deferred bugs are tagged with the suffix " `.deferred.<bug-number>`" on their file name. (`<bug-number>` is of course the actual bug number from the bug database.) Deferral of the tests is an interesting concept, and a test may be deferred for two reasons:

1. The test may be deferred because of an outstanding bug in the bug database. All bugs are tagged with the number of the requirement that is unfulfilled by the bug itself. If test 2.4.6.7.8.2 has an outstanding bug reported against it, there is no sense in running the test again until the bug is fixed. Indeed, it is one less test exception that the test analyst has to wrestle with, and this facility also helps prevent multiple submissions of the same bug reports.
2. The test may be deferred because of another outstanding bug that is wreaking havoc with the test suites in general. Consider the case in which an outstanding bug makes it impossible to get a reliable status report from the printer. This would make it desirable to remove any tests that rely upon a status report from the test suite. Any test may be deferred by placing two fields in a special file used by the `tag_deferred_tests` script: the number of the test to defer, and the bug number of the bug that is responsible for its deferral.

The next step is building the test suites into logical subsuites that will be run on the test harness as individual runs. This grouping is accomplished by a shell script called `block_suites`. This script simply searches each of the directories mentioned above and concatenates all tests in a particular section together. In other words, we happen to know that all requirements in chapter 2.4.1 are requirements dealing with software protocols. It would make sense, then, to collect all files whose name begins with the string '2.4.1' into a single suite and give it appropriate name like: `swproto.for`. The `.for` extension on the file name in this example alludes to the fact that this is FORTH source that may be directly executed on the test harness described in the next

section.

The final step is to collect any metrics on the test suites in general. These metrics are used primarily to monitor the progress of the test writing effort. The shell script `count_suite` produces a report in a file called `count` in the current directory. Here is an actual example of the report produced early in the test harness development cycle:

```
Requirements that have....

    Automatic tests:            633     (  44.5 % )
    Manual tests:               190     (  13.3 % )
    No tests written:           320     (  22.5 % )
    Partial tests written:      277     (  19.5 % )
    Total requirements:         1420

Total number of tests in all suites:   1430
```

This report is produced by simply counting the number of instances of the `TS` and `RQ` operators in the directories. Keep in mind that a requirement may have more than one test.

3.5 Automated testing tool (MOTHER)

The automated testing tool is the physical manifestation of the MOTHER system, and is the portion of the system most frequently referred to as "MOTHER".

The testing tool consists of several elements:

1. The modified FORTH interpreter.
2. The PC-NFS file system link to the server.
3. The test monitor tool.
4. The automated test exception report tool.

Figure 1 shows how the fundamental elements of the PC computer interface with the system under test.

3.5.1 The FORTH interpreter

At the heart of the automated test harness is a public domain FORTH interpreter, written entirely in "C". We have modified this FORTH interpreter to compile under the Turbo-C++ compiler on a PC computer.

We installed a set of custom operators in the FORTH interpreter to allow us to manipulate a set of off-the-shelf instrumentation boards that allow us to drive the printer firmware inputs, and monitor printer firmware outputs.

The FORTH interpreter also has a set of operators that interface with commercially available RS-232 drivers which allows MOTHER to communicate with the diagnostic interface in the printer.

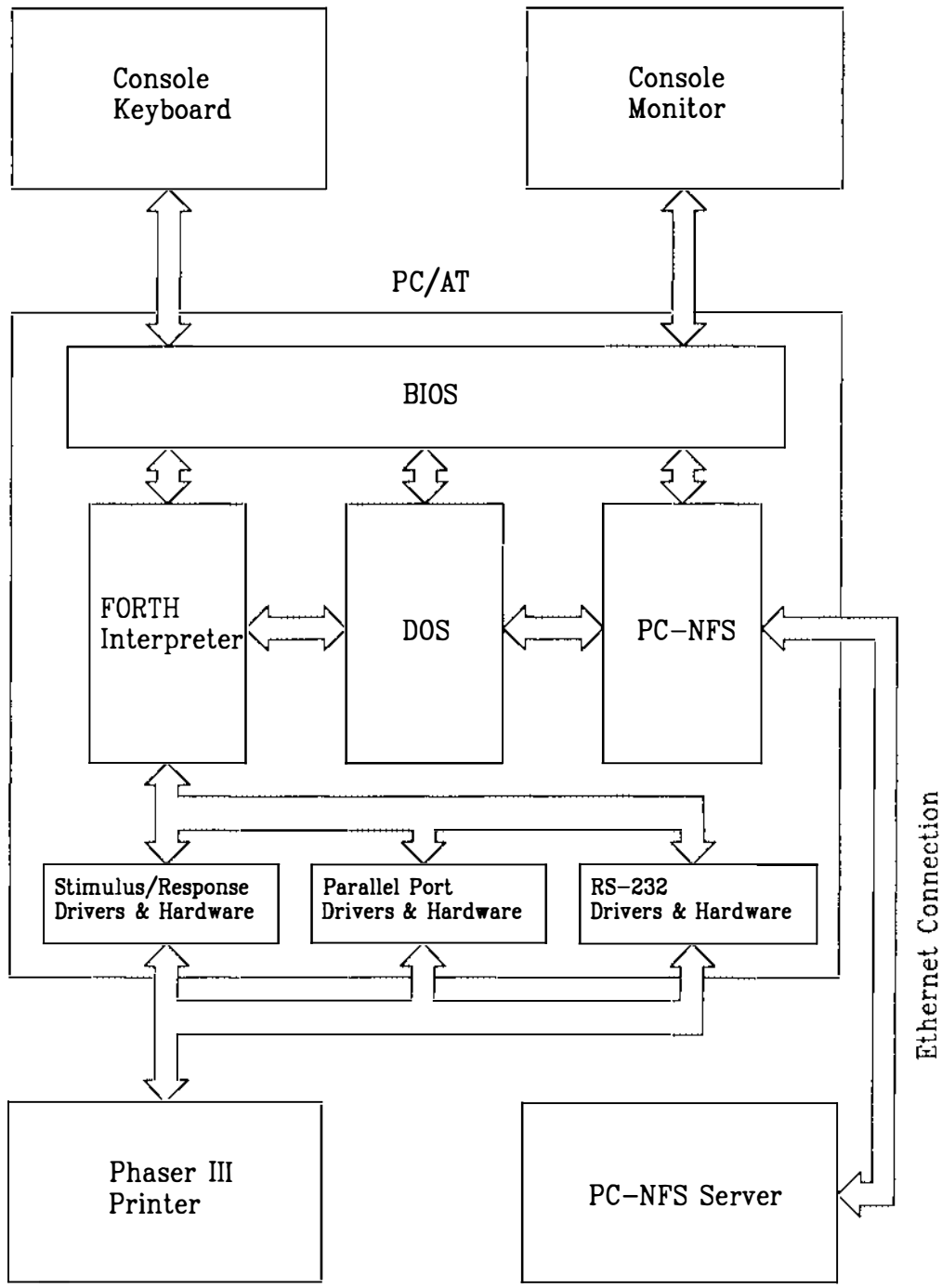


Figure 1: Test Harness Architecture

3.5.1.1 Custom test operators

Custom test operators were constructed to interface the test stream with the test hardware, specifically firmware inputs and outputs. These custom test operators were constructed on a three-tier scheme. The three tier scheme was as follows:

1. Operators could be coded in FORTH, as an aggregate of other FORTH operators.
2. Operators could be coded in C, with interfaces to FORTH level calling styles (i.e. parameters could be passed on the FORTH stack).
3. Operators could be coded in C, with interfaces to other C routines. (i.e. parameters could be passed on the processor stack).

This scheme provided engineers with the ability to interface C with FORTH and FORTH with C. It also allowed engineers to construct prototypes of the operators very quickly, using the FORTH interpreter itself.

Experience indicates that the C interface with FORTH operators was never used: engineers worked from the FORTH level down to the C level, and never in the other direction. This would be in keeping with normal top-down programming techniques.

3.5.1.2 Custom hardware interface

The custom hardware interface was a set of off-the-shelf boards that were interfaced with the printer under test. The overall cost of the off-the-shelf boards for each test harness was approximately $1000. At the time the test system was designed, the firmware environment had already been defined, and the hardware interface required some modification of stock hardware to support the test harness.

One of the requirements of the test harness is that there could be no modification of the system under test that affected the firmware. The reason for this is obvious: **if you are testing a custom version of the firmware, you are not testing the *customer's version* of the firmware.**

The hardware was constructed using a *stimulus/release* approach. This approach provides a mechanism which allows the printers native hardware to assert normal operating conditions to the firmware when not overridden by the test harness. To illustrate this concept, consider the following example:

It is desirable to force certain ink-level conditions in the course of testing a printer. We may want to force ink-levels that are *FULL, HALF,* or *EMPTY* at the firmware interface. *However, when we are not providing stimulus to the ink-level inputs of the firmware, we want the ink-levels to register their real values.* This is so that in the course of testing that doesn't exercise the ink-levels as part of the suite, the operator will have some indication of whether additional ink is required for normal operation of the printer. (This is especially true when running a suite of tests that makes a lot of prints.) Therefore, in addition to a hardware interface to the ink-levels that provides the *FULL, HALF* and *EMPTY* stimulus, we must also provide a *RELEASE* operation that allows the hardware to detect the normal (true) levels of ink.

All hardware interface operators have a *RELEASE* operation, and all hardware interfaces are *RELEASED* at the end of each test. All interfaces are *RELEASED* at the end of each test, so that it is not necessary for the test writers to clean up at the end of each test: the test writers can simply provide the necessary stimulus, check for the results, and leave the system in its current state. The cleanup is automatic, and is an integral part of the `TE` operator.

3.5.2 The PC-NFS file system

The PC-NFS file system is used to run tests and store test logs directly in the test suite directories. The test suite directories are generated by the server and the server structures are mounted on the PC computer.

As a consequence, when tests are run, the executable image of the MOTHER automated test system and the source suites are actually on the server. The automated test harness writes its log files directly to the PC-NFS mounted directories, and no files are used from the PC. Source files for the MOTHER automated test harness are compiled on the PC and use PC local directories (as opposed to the PC-NFS directories) for development purposes: licensing considerations dictated that our commercial libraries and packages cannot be shared as freely as source files, so we were obliged to keep the development structures local to the PC machine itself.

3.5.3 Test monitor tool

Since the PC-NFS server has both the test suites *and* the test logs available, it is possible to monitor the progress of the suites by comparing the actual suites with the resultant audit trail (test logs) on the server. A utility on the server, `pct_done`, does this comparison and prints the percentage of the testing that is currently done. By using this facility, the test technician can estimate the amount of time left in the execution of a particular suite.

3.6 Automated test exception report tool

Immediately after a particular suite has executed, an automated test exception report tool may be used to generate exception reports. This tool, `failure_reports`, is a shell script that uses simple utilities to generate the exception reports from the test report logs.

3.6.1 Test report logs

The automated test harness writes several error logs during execution of the test suites:

1. `err.log`: The log file of errors detected by the test harness.
2. `mother.log`: The log file of the executed test operators.
3. `zterm.log`: The log file of data received from the RS-232 diagnostic interface (terminal interface).
4. `instr.log`: The log file of instructions issued to the operator during the run of the test suite, and the operator responses to those instructions.
5. `req.log`: The log file consisting of the actual text of the requirements that were displayed during the test. (Requirements are displayed upon the console while the test for that requirement is actually being performed.)

3.6.1.1 Format of the log files

Every time a record is written to a log file, five essential elements are written:

1. *The sequence number.* This is a number that is incremented every time *any* record is written. This number is the first element written to a file, and is a fixed format integer, occupying six columns in our implementation.
2. *The time stamp.* This is primarily to give the test analyst an idea of relative time frames involved. It is also a fixed format field, consisting of hour, minute, second and millisecond.
3. *The number of the requirement under test.* The custom FORTH operator `RQ` places this information in a global variable so that this information is available to all routines within the test harness.
4. *Tag indicating the file that the record was written to.* This tag is an "E" for `err.log`, "M" for `mother.log`, "Z" for `zterm.log`, "I" for `instr.log`, and "R" for `req.log`.
5. *The message.* The remaining information is simply the content of the message written to the specified file.

The following is a brief excerpt from the `mother.log` file of the *ready mode* test suite that illustrates the use of this format:

```
000005 16:44:06.45                  -M- ----Stream opened----
000006 16:44:06.67                  -M-  [ 0 ] OK
000007 16:44:06.78                  -M- RQ  [ 0 ] OK
000014 16:44:07.44  2.4.3.10.0.0.1  -M- TS  [ 1 ] OK
000015 16:44:07.55  2.4.3.10.0.0.1  -M- SET-READY-MODE  [ 6 ] OK
000018 16:44:09.03  2.4.3.10.0.0.1  -M-   [ 6 ] OK
000019 16:44:09.14  2.4.3.10.0.0.1  -M- FULL BLACK INK-LEVEL  [ 7 ] OK
000020 16:44:09.53  2.4.3.10.0.0.1  -M- LOAD-MEDIA ( set ink levels )  [ 9 ] OK
000023 16:44:25.18  2.4.3.10.0.0.1  -M- FORM-FEED  [ 11 ] OK
000029 16:44:46.88  2.4.3.10.0.0.1  -M-   [ 11 ] OK
000030 16:44:47.04  2.4.3.10.0.0.1  -M- DECIMAL 3 SECONDS SET-DEADMAN-TIMER  [ 12 ] OK

etc.
```

The numbers in brackets indicate the depth of the FORTH stack at the time the execution of a line is completed. The `TE` operator will examine the FORTH stack at the end of a test, and print out the depth at which any "FAILED" or "TIMED OUT" markers have been placed on the stack. This makes it simple to locate which part of the test failed, and it also makes it simple to determine the reason for this failure.

The power of this elaborate format will become apparent in our discussion of the test exception report generator in the next section.

3.6.2 Shell script: failure_reports

The `failure_reports` utility extracts lines from the error log that contain the keywords *FAILED* or *TIMED OUT*. Since every line in the error log is tagged with the requirement number, it can extract the actual requirement number from the tagged line. These requirement numbers can then be sorted and piped through a filter designed to ensure that each requirement number appears only once. (This prevents duplicate exception reports, should a particular requirement have more than one failure entered in the error log.)

At this step in the script, we have a unique list of requirements whose tests have logged at least one failure. The next step is to extract the lines from the other relevant files to produce a coherent picture of the actual sequence of transactions that took place. The experience of the test analysts indicate that to form this coherent picture of the actual failure, we need information from the requirements text log (`req.log`), the error log (`err.log`), the diagnostic RS-232 port log (`zterm.log`), and the test log (`mother.log`).

For each requirement in turn, the script:

1. Uses `grep` to extract all lines pertaining to that requirement from the files indicated.
2. Pipes the `grep` output into `sort`, which sorts by the first field (the sequence number).
3. Pipes the `sort` through `pr` to format an exception report listing conducive to analysis. Thus, each page of the exception report has a heading line indicating requirement number, time of generation, etc.

Here is an example of an exception report listing:[11]

```
Jun  4 12:59 1991  Requirement 2.4.3.10.0.0.2                Page 1

Requirement: 2.4.3.10.0.0.2

000067 16:45:28.07  2.4.3.10.0.0.2  -R- Requirement 2.4.3.10.0.0.2
000068 16:45:28.18  2.4.3.10.0.0.2  -R-
000069 16:45:28.18  2.4.3.10.0.0.2  -R- [Window cleared]
000070 16:45:28.29  2.4.3.10.0.0.2  -R- The Ready mode (Ready--), shall proceed to Ink Load mode
000071 16:45:28.34  2.4.3.10.0.0.2  -R- lever), within 2 seconds when a black ink in preload
000072 16:45:28.45  2.4.3.10.0.0.2  -R- position event occurs and the black ink level is low.
000073 16:45:28.62  2.4.3.10.0.0.2  -M- TS  [ 1 ] OK
000074 16:45:28.78  2.4.3.10.0.0.2  -M- SET-READY-MODE  [ 6 ] OK
000075 16:45:30.10  2.4.3.10.0.0.2  -Z- [Ready--]
000076 16:45:30.21  2.4.3.10.0.0.2  -M-   [ 6 ] OK
000077 16:45:30.32  2.4.3.10.0.0.2  -M- HALF BLACK INK-LEVEL  [ 7 ] OK
000078 16:45:30.71  2.4.3.10.0.0.2  -M- LOAD-MEDIA ( set ink levels )  [ 9 ] OK
000079 16:45:46.19  2.4.3.10.0.0.2  -Z- [Printing--]
000080 16:45:46.25  2.4.3.10.0.0.2  -Z- HH: 98
000081 16:45:46.47  2.4.3.10.0.0.2  -M- FORM-FEED  [ 11 ] OK
000082 16:46:07.62  2.4.3.10.0.0.2  -Z-     654    710    837   4126
000083 16:46:07.73  2.4.3.10.0.0.2  -Z- HE: 709 LE: 837 TE: 4126 L: 3289
000084 16:46:07.84  2.4.3.10.0.0.2  -Z- TEND: 4666 TS: 4682 Rem C: 8
000085 16:46:07.95  2.4.3.10.0.0.2  -Z- Park: 5892
000086 16:46:08.00  2.4.3.10.0.0.2  -Z- [Ready--]
000087 16:46:08.22  2.4.3.10.0.0.2  -M-   [ 11 ] OK
000088 16:46:08.33  2.4.3.10.0.0.2  -M- DECIMAL 3 SECONDS SET-DEADMAN-TIMER  [ 12 ] OK
000089 16:46:08.88  2.4.3.10.0.0.2  -M- READY WAITFOR-FRONT-PANEL-MESSASGE WAITFOR-FRONT-PANEL-MESSASGE?
000090 16:46:09.37  2.4.3.10.0.0.2  -M-   [ 0 ] OK
000091 16:46:09.48  2.4.3.10.0.0.2  -M- ASSERT BLACK INK-IN-PRELOAD-POSITION  [ 1 ] OK
000092 16:46:10.09  2.4.3.10.0.0.2  -M-   [ 1 ] OK
000093 16:46:10.14  2.4.3.10.0.0.2  -Z- BLACK
000094 16:46:10.31  2.4.3.10.0.0.2  -M- DECIMAL 3 SECONDS SET-DEADMAN-TIMER  [ 2 ] OK
000095 16:46:10.69  2.4.3.10.0.0.2  -Z- [Pull ink load lever]
000096 16:46:11.08  2.4.3.10.0.0.2  -M- PULL_LEVER WAITFOR-FRONT-PANEL-MESSASGE WAITFOR-FRONT-PANEL-MESSASGE?
000097 16:46:11.68  2.4.3.10.0.0.2  -M-   [ 0 ] OK
000098 16:46:11.79  2.4.3.10.0.0.2  -M- ?INK-SUPPLY-LOW FALSE-IS-SUCCESS  [ 2 ] OK
000099 16:46:12.34  2.4.3.10.0.0.2  -M- ?INK-LOAD-IN-PROCESS TRUE-IS-SUCCESS  [ 3 ] OK
000100 16:46:12.89  2.4.3.10.0.0.2  -M-   [ 3 ] OK
000101 16:46:13.00  2.4.3.10.0.0.2  -M- ?FAULTED TRUE-IS-SUCCESS  [ 4 ] OK
000102 16:46:14.32  2.4.3.10.0.0.2  -M- TE  [ 0 ] OK
000121 16:46:43.10  2.4.3.10.0.0.2  -E- 2.4.3.10.0.0.2 Test 1 operation 2 FAILED
000122 16:46:45.24  2.4.3.10.0.0.2  -Z-
000123 16:46:45.29  2.4.3.10.0.0.2  -Z- [Ready--]
```

In this particular example, the test failed due to sloppy typing by a test engineer that misspelled *MESSAGE* as *MESSASGE*, indicated on lines 89 and 96.

[11] The author regrets to disclose that this exception report listing has been subjected to a little prudent censorship to prevent confusion. At the end of this test, the printer was reset as a result of the failure. The printer is reset whenever a failure occurs, to ensure that it (the printer) is in a quiescent state before the next test begins. The power-up messages contain abbreviated diagnostic messages that are only coherent to the *illuminati* of the design team. Even I couldn't explain the messages contained therein, hence I am reluctant to print examples that defy a complete explanation.

4. Using MOTHER

MOTHER is designed to be simple to use. The environment is geared toward speed and response time. This section describes a simple method for using MOTHER that highlights its flexibility.

4.1 Generating the Software Requirements Specification

To generate the *Software Requirements Specification* (SRS), one need only do the following things:

1. Choose a template, and setup a skeleton of the SRS document. We chose to organize our document along NASA's SFW-DID-08.[12]
2. Write the requirements, delimited in the source files by the `.RQ` and `.RE` macros.
3. At this stage, one *could* begin writing the tests. May I be so bold as to suggest that it would be prudent to subject the document to review first? This precaution may prevent effort from being wasted by writing tests for requirements that may be changed extensively.
4. Write the tests, delimited in the source files by the `.TS` and `.TE` macros.
5. Review the tests.
6. Run and correct the tests.

The method for generating the documents for the above steps is described in the following sections.

4.1.1 Generating the Software Requirements Document

To generate the SRS, a special switch is provided in the body of the document to turn the "*print tests*" on or off. If no tests are written for the first review of the requirements then the switch is not of immediate importance.

For later use, the switch will be set to the "on" position to generate copious quantities of documentation, showing the tests *in context* immediately next to their requirements.

4.1.2 Generating an SRS listing for review of requirements

The following is a brief example of the format of the document when generated with requirements alone:

[12] I have never actually *seen* this specification. I ran across the outline in a book on writing specifications, and asked our *Technical Standards* folks to locate it for me. I continued writing from the outline of the standard in the book while the Standards folks began what became a *Quest for the Grail*, so to speak. They couldn't locate anyone at *any* NASA facility who had heard of this. Since our *Technical Standards* folks took this as a matter of professional pride, they wrote to the publisher who forwarded their request to the author. The response from the author was essentially "*never mind*": the author indicated simply that the NASA requirement would be dropped from the next revision of the book. I wonder if it ever existed at all? It looked nice, but the truth of the matter is that our SRS may be based on a non-existent standard. Although the standard may be non-existent, the format served us very well.

```
Requirement 2.4.1.2.0.0.0.4

The ready mode (Ready--), shall proceed to the power up mode
(Busy--Self Test), within 1 seconds following the release of
the INPUT PRIME line.

Test Description

        Sequence -      (set ready mode)
                        (assert input prime)
                        (check for power up mode)
        Results -       Mode changed to power up within 1 sec.
```

As can be seen, a summary of the requirement and a simple, straightforward description of the proposed test(s) are produced together. This simple description of the proposed test allows the less technically inclined to see the nature of the requirements testing, and allows opportunity for comment.

4.1.3 Generating an SRS listing for review of tests

Where the SRS is to be reviewed from a QA engineering standpoint, a copy must be generated that includes the actual test itself, as in the following example:

```
Requirement 2.4.1.2.0.0.0.4

The ready mode (Ready--), shall proceed to the power up mode
(Busy--Self Test), within 1 seconds following the release of
the INPUT PRIME line.

Test Description

        Sequence -      (set ready mode)
                        (assert input prime)
                        (check for power up mode)
        Results -       Mode changed to power up within 1 sec.

  Test:

              SET-READY-MODE
              RESET
              DECIMAL 1 SECONDS SET-DEADMAN-TIMER
              WAITFOR-FRONT-PANEL-DISPLAY Busy--Self test      "
              ?SELECT FALSE-IS-SUCCESS
```

The resulting document may be reviewed by test engineers to ensure that the tests do indeed test the indicated requirement.

4.2 Generating the test suites

Test suites are generated by using the `build_test_suite` utility mentioned in the previous section entitled *What is MOTHER?*

4.2.1 Manual generation

The test suite may be generated on demand by creating an empty directory and running the `build_test_suite` utility as mentioned in the previous section. The utility itself generates a test log, and, in our case, with about 1400 to 1500 requirements and their tests, it requires about 55 minutes to an hour to generate on an unloaded VAX 8650.

4.2.2 Automated generation using the "at" utility

We have found it useful to generate a test suite using the `at` utility on our VAX. Every morning at about 1 a.m., when the VAX load factor is low,[13] I run jobs that generate a new set of test suites (in case we need them), and a printer-ready copy of the SRS.

4.2.3 Inspecting the results

The `build_test_suite` utility generates output, which I usually redirect to a file called `build.log`. This audit trail will register any problems or fatal errors that are encountered while building the test suites.

Here is an edited sample of what the log file looks like:

```
Tue Jun 04 1991 (01:04:43 AM) Expanding SRS.  Please wait.
Tue Jun 04 1991 (01:06:02 AM) Extracting tests.  Please wait.
Tue Jun 04 1991 (01:10:15 AM) Moving tests into automatic and manual test directories.  Please wait.
Tue Jun 04 1991 (01:10:22 AM) Moving test files containing partial tests
                                  :
                                  :
                                  :
Tue Jun 04 1991 (01:21:50 AM) Moving test files containing WAITFOR-DRUM-HOME
Tue Jun 04 1991 (01:22:29 AM) Moving test files containing WAITFOR-DRUM-MOTION
Tue Jun 04 1991 (01:23:05 AM) Moving test files containing WAITFOR-MEDIA-SCAN
                                  :
                                  :
                                  :
Tue Jun 04 1991 (01:49:31 AM) Marking tests to be deferred.
Tue Jun 04 1991 (01:49:34 AM) Blocking suites.
Tue Jun 04 1991 (01:51:07 AM) Counting the suites.
Tue Jun 04 1991 (01:51:31 AM) Cleaning up.  Please wait.
Tue Jun 04 1991 (01:51:31 AM) Done.
```

The readers who have read the section entitled *What is MOTHER?* will recognize the messages are from the individual utilities that were used at each step of the `build_test_suite` log.

4.3 Running the test suites

Once the test suites are generated, the running of the test suites is simple. From the PC, one need only change to the directory that contains the test suite that you care to run. Then it is a simple redirection of the file. For example, to run the test suite `ready.for` from the console, the test technician only needs to type:

```
mother < ready.for
```

The automated test harness will then provide the rest of the directions, if any, for the test operator.

4.3.1 Automated tests

Once the test harness has been started, tests that have been sorted into the automated test directory can be run without human intervention.[14] The fully automated tests may be started, and essentially ignored until they run to completion.

[13] This is not to imply it is unused. I have received electronic *hate mail* from folks who are working at 1 a.m. They were angry because my jobs ran the load factor up for about an hour.

[14] Yet another great lie. For printers to operate properly, they require a sufficient amount of ink and paper. When they run out, the test harness will stop, request a refill of the exhausted resource and patiently wait for the operator to comply. (There are two great advantages to computers: they perform exhausting and repetitive tasks without complaining, and they wait patiently for the test engineer to get back from coffee break.)

4.3.2 Manual tests

Upon occasion, it is necessary to ask the human operator to do what the harness cannot do: annotate test prints with required information (you simply can't trust the printer to do the annotation itself), load a special size of paper for the test (such as B-size or a metric-sized paper), or any number of other manual operations.

These tests are sorted into a directory of manual tests, and require close operator attention to run. They are sorted into the manual test directory by means of keywords placed in a special file, as described in the previous section *What is MOTHER?*

4.4 Analyzing the output of the test suites

After the run of a particular test suite, analysis may begin immediately. Bugs from the test suite fall into three main categories:

1. Bugs in the printer firmware.
2. Bugs in the test suite (misspellings, etc.).
3. Bugs in the automated test facility itself, such as bugs in the FORTH operators.

The classification of each bug into its correct category is of vital importance to maintaining a healthy relationship with the design team. This is accomplished by:

1. Generating the test exception reports.
2. Inspecting the test exception reports carefully, and assigning bug classifications.
3. Reproducing bugs to be sure that the classifications are reasonable.
4. Submitting the error reports to the design team.

4.4.1 Generating the test exception reports

The test exception report is generated by using the `failure_report` shell script described in the previous section. It generates printer ready audit trails of the tests that failed.

4.4.2 Inspecting the test exception reports

These exception reports are then inspected by the test analysts. The test analyst is the most important element in the test harness operation. We have discovered that, in many cases, the test suites run faster than the test analyst can analyze the results!

The test analyst is important because often the test analyst can spot mistakes in tests at a glance. Errors in this category are usually misspellings or misuse of operators. These mistakes are often very consistent, or "systematic". Often, systematic errors in the test suites can be quickly corrected, and the test suite can be run again in the same QA cycle.

4.4.3 Reproducing bugs and submitting error reports

The next stage is for the test analyst to reproduce the bugs by manually stepping through the test suite to ensure that the exception is not dependent upon abnormal conditions. This is done by simply bringing up MOTHER without redirecting standard input as described previously in the section *Running the test suites.* This allows the test analyst to reproduce the bug by typing the operations manually, and to experiment with the condition in hopes of gaining additional information for the design engineering team.

4.4.4 Submitting error reports

In the final stage of the cycle, the QA test analyst can include the actual text of the test in the bug report, so that the bug database carries the actual *reproducible test* for generating the exception. This provides design engineering with an actual plug-and-play bug generation mechanism that cuts the time required to reproduce the bug.

When regression testing is done, the ''bug closure'' procedure already has the test in place for testing. This shortens the time for regression testing, as a single suite may be synthesized with tests for all of the supposedly ''fixed'' bugs that can be tested as a monolithic run, if desired.

5. Experiences with MOTHER

5.1 A test harness of this nature is a complete project itself

When planning to build a test harness, it is important to acknowledge that a test harness is a project itself. Basically, there are two things to consider:

1. Test harnesses need project planning, too.
2. Test harnesses need design evaluation as well.

5.1.1 Test harnesses need project planning, too

One of our major oversights was the time required for the development of the test harness. Seldom is sufficient analysis and time given to the correct scheduling of supplemental programs and harnesses for testing: it is at least as much of an oversight in QA efforts as in design engineering (perhaps more so).

The reader is strongly advised not to oversimplify the problem of a project within a project. There are several essential planning elements to consider when building a test harness. These elements are:

1. Manpower requirements.
2. Schedule requirements.
3. Performance requirements.
4. Equipment requirements.

All of these requirements hinge upon the definition of the test harness itself, forcing us to deal with a true paradox:

> *How do we define the requirements for a test harness whose fundamental requirement is to be flexible in its requirements?*

With no experience base to work from, this was enough of a problem to prohibit any worthwhile estimation. Now that we have the figures, which we will share in the next section, the problem is a little less paradoxical.

5.1.2 Test harnesses need design evaluation as well

We have already stated that the test harness is a project within a project. This implies that, as in the case of a commercial project, you have a simple choice: either you evaluate the design of your product or the customer will. In this case the customer is the combined design engineering and QA groups.

An evaluation phase is almost always a painful one for the design engineers. In this instance, the design engineers and the QA engineers are one and the same. It is important that during the design evaluation phase, a certain amount of visibility of this process is given to the design engineering team. There are two fundamental reasons for this:

1. In the future, the design engineers will be asked to accept the diagnosis provided by the harness. It is imperative, therefore, that the design team have confidence in the design of the test harness.
2. In the future, the design engineers will be also subjected to the criticism inherent in the QA team's evaluation of their code. It is imperative, therefore, that the design team does not feel that they are being required to follow design standards that are more rigorous than those followed by the QA team.

5.1.3 We can learn from the implementation of the test harness as well

Formal development of a test harness provides a unique opportunity for the QA community in a company as a whole: it provides a baseline for estimating the effort of future test harness development.

In this particular effort, the QA team tried to capture as much information as possible. Since this effort was, for the most part, a different approach than what is usually used, it wasn't apparent what metrics should be collected. Fortunately, the metrics we did collect seem to be sufficient to answer most questions about developing and scheduling test harnesses of this genre.

5.2 What worked well?

Many things worked well with our test harness: some things worked much better than expected.

5.2.1 Timely turnaround was achieved

The original QA plan called for *three* test harnesses and *five* people. We ended up with *one* test harness and *three* people, myself included. In spite of this, the target two day turnaround was achieved. If we had had the equipment we originally wanted, we could have exceeded this goal, perhaps providing *one day* turnaround. We needed to run three shifts (around the clock) to accomplish the two day turnaround time, but it *was* accomplished. [15]

We were able to scan tests and provide feedback to the test writers in a matter of hours. The longest test suite was about five hours long.[16] The average test suite run time was two to three hours long. In this respect, we satisfied the requirement that we provide fast feedback to the test writers as to the correctness of their tests.

Additions of new test operators were typically accomplished in a matter of an hour or two per operator. Some new operators were installed as a composition of FORTH operators. In these cases, the addition of new operators was typically accomplished literally in a matter of minutes.

[15] I would not advise anyone to try a single-unit approach as described here. We were required to use a single-unit approach because our *Phaser III Printer* prototypes were expensive and extremely hard to come by. The risk of having a single test harness that may be subject to breakdown at inopportune moments makes a single harness approach less than desirable. Also, running around the clock shifts causes extremely high levels of stress among the test technicians. Our test technicians went to heroic lengths to accomplish the two day turnaround.

[16] This is an absolute lie. The longest test suite was 11 hours long until we fixed it so that it worked as it should. A certain amount of honesty seems in order here. If a test suite took more than a couple of hours to run, we did a close analysis to determine why it took so long. In every instance the inordinately long test suites took so long because of some overlooked or unimplemented functionality in the test harness.

5.2.2 Firmware defects in the product under test were exposed

In the course of installing the test harness and running tests against the prototype *Phaser III Printer*, we systematically exposed latent defects in the existing firmware. In short, because we were using the printer in a manner that was never anticipated, we exposed certain defects in the *Phaser III* that may have rarely been seen in everyday use. In some cases, this blatant "misuse" of the printer may have exposed defects that were lurking in the fringes of the firmware architecture itself.

5.2.3 The test harness was "accepted" by the design team

The acceptance of the test harness by the design team was immediate. Although the test harness was released to the design team at a reasonably late time in the project, the ability of the harness to reproduce failure conditions *delighted* the main design engineer.

5.3 What could have worked better?

As is the case with every project, there were instances that left less than desirable conditions for the evaluation of the test subject. In each of these instances, we needed to provide an alternate mechanism for bridging the gap left by inadequacies in the harness.[17]

5.3.1 Hardware interface should have been designed into product

The hardware interface between the test harness and the product was a wiring nightmare. In this product, the QA team arrived late on the scene, and had no input into the hardware design. It is no surprise, therefore, that the hardware was not designed with firmware testability in mind. Since we wanted to use a *state-stimulus-response* approach, it was necessary that the test harness have access to all *inputs and outputs* used by the firmware. Since access to all firmware input and outputs was necessary, we were forced to actually *disembowel* the printer to gain access to these signals.[18]

5.3.2 Granularity of time domain for test was less than desired

The PC/AT provides a less than ideal time base for our tests. In this instance, we ran our time domain tests in an environment where we had an eighteenth of a second resolution of time events. Our test harness took advantage of the system timer tick, and the test harness code intercepted the timer tick interrupt to set a time base for our *deadman timer* facility. Games could indeed be played with reprogramming the system timer chip and passing a reduced number of interrupts through to the operating system. However, there is another barrier lurking in this scheme: the basic instruction cycle time. There is a great deal of potential for marrying the test harness with a programmable logic analyzer for high speed time-domain measurements. This approach is more promising for solving time-domain issues.

5.3.3 Number of test operators was large and difficult to remember

If you make it possible for people to write a new test operator at a moment's notice, you will find that people will write new test operators at a moment's notice.

---

[17] This is the real meat of the experience from our perspective. This section is an explicit list of what we will improve upon the next time.

[18] This has its advantages: no one asked to borrow a printer that looked like it was completely disassembled.

The number of test operators was extremely large and soon became somewhat cumbersome. It is doubtful that *anyone* was completely conversant in the entire test description language that resulted. The test writers developed their own test operators that were prose-like and very descriptive, while the test analysts developed their own shorthand that made the same operator easy to type. For instance, the operator `GET-ERRORS-REPORT` was shortened to `GER` by test analysts, yielding two operators that did the exact same thing. While the functionality of the first is readily apparent in its verbose name, the cryptic shorthand notation is most usable by the "two-fingered" test analyst who is trying to reproduce the failure of a somewhat lengthy test. Anyone who uses the approach outlined in this paper may want to ensure that their test analysts are capable of touch-typing.

5.3.4 Could have used more time to train test writers

We certainly could have used more time to train the test writers. (In fact having actual test *writers* would have been an advantage: we only had a test *writer*.) The time required to train the test writers would have been spent primarily on providing the writing team with better documentation dealing with:

1. A list of operators grouped by functionality.
2. A list of operators with FORTH's so-called *stack pictures*: a list of what needed to be on the stack *before* the call, and a list of what was left *after* the call.
3. A short paper on the philosophy of the test harness.
4. A few real examples of various types of tests.

These four simple items could have saved many hours for our test writer.

5.3.5 The test procedure was horribly tedious

If you make a test harness that even the simple-minded can operate, then you should hire the simple minded to operate it. The use of the technically literate to perform this extremely mundane task was an inhumane torture. The actual procedure of testing was horribly tedious.[19]

5.4 A test harness can have lasting value

5.4.1 A demonstrated method

We now have a demonstrated method for integrating test and specifications while maintaining flexibility. This is of particular interest to project managers, since it provides for the flexibility of the entire product. This strategy provides for an extremely late binding time for the specification, and this is of the utmost importance for a project such as the *Phaser III*, where simple changes to a single element can ripple through the entire system.

[19] I'm bragging here: It's one of my fundamental beliefs that when engineering gets exciting, it gets exciting in a *bad way*. It is, consequently, a measure of success that this testing process was boring: no ugly surprises. This test harness performed over 1,400 tests with clockwork regularity and consistency. It would be a crime against humanity to require a human to perform the 1,400+ tests with the same accuracy and consistency. Our test technicians listened to radio and played computer games during the periods when the fully automated tests were running: they only had to check on the harness every five minutes or so to ensure that everything was running correctly. In most automated test suites, the *Phaser III* would make enough noise loading paper that it was clear that the system was operating correctly, and only extended periods of silence from the test harness aroused suspicion.

5.4.2 It can only get better

We have provided an analysis of the entire effort showing the areas for improvement, and describing exactly *what* can be done to improve these areas. None of the areas impacted the quality of the product, but rather these areas were primarily associated with making the entire test harness more useful to the customers: the design engineering and QA groups.

5.4.3 A springboard for the next harness

For future products, the test harness as it now exists will cut down on the development effort for the next generation of harnesses. Since all improvements were incorporated in the harness as we went along, it seems reasonable to claim that all the server-based tools are 100% reusable. The actual off-the-shelf boards are, of course, reusable. For any new project:

1. The product-specific test operators will need to be changed.
2. The hardware interface should be cleaned up. (Perhaps a single test harness interface connection to the product under test is in order.)
3. The *Software Requirements Specification* (SRS) will need to be written for the new product.
4. New tests will need to be written for the requirements in the new SRS.

In terms of expenses, there are no new capital expenses for the test harness itself: all new expenses are manpower expenses. All hardware and a large portion of off-the-shelf software[20] are reclaimed.

[20] Compilers, shell scripts and the like. This does not include the test operators themselves, although a large number of them may be reclaimed as well.

6. Conclusion

6.1 Presentation of metrics and numbers

6.1.1 Time to develop test harness

Let me begin by giving a feel for the size of the software in the test harness itself (the PC/AT-based software):

```
Number of lines of code:        9260
Number of blank lines:          4656
Number of comments lines:      11983
```

Applying these numbers to Boehms *cocomo* [21] estimation model, we would project a development schedule as follows:

```
Model mode: organic
Model size: intermediate (9260 lines of code)

Total effort:   24.8 man-months       [152 man-hours/man-month]
Total schedule:  8.5 months           [standard calendar months]

Distributions:                  Effort          Schedule        Personnel
                              (man-months)      (months)        (on-board)

    Plans and requirements: (06%)  1.5     (11%)  0.9          1.6
  Product design:           (16%)  4.0     (19%)  1.6          2.5
  Programming:              (65%)  16.1    (59%)  5.0          3.2
    Detailed design:        (25%)  6.2
    Code and unit test:     (40%)  9.9
  Integration and test:     (19%)  4.7     (22%)  1.9          2.5

Programmer productivity during code and unit test phase:  932 DSI/month.
[DSI = Delivered Source Instructions]
```

In actuality, this harness was developed in two months by two engineers. Needless to say, we worked twelve-hour days and most weekends. The implications are that there was a productivity gain of 6.2 *over* the standard man-month. We did not leverage off of any new technology, we just worked *very* hard. Certain productivity gains can be directly attributed to the modularity of the test harness itself: developing lots of little operators is much like having lots of little projects. The FORTH interpreter itself consists of 1031 lines of C code, and has been included in the above code counts for the sake of simplicity.

[21] Reference: Boehm, Barry W. *Software Engineering Economics*, Prentice-Hall, New Jersey, 1981.

6.1.2 Time to develop tests

This becomes somewhat of a problem, since the development of tests is an ongoing effort: things were still changing near the end.

We have, at last count, in our SRS (including tests):

```
Number of lines:            65840
Number of ''words'':       178583
Number of characters:     1350606
Number of requirements:      1429
Number of tests:             1473

Percentage of tests requiring manual intervention:   7.9%
```

6.1.3 Time to generate test suites

The generation of test suites took 41 minutes on an unloaded VAX 8650. In addition, it took 2.7 megabytes of disk space to generate the test suites (without the associated log files).

6.1.4 Time to run test suites

There were nine sub-suites that could be run independently. The average time for each of these sub-suites was about 3 hours, although the longest took 5 hours. The total time to run all nine automatic test suites was twenty-seven hours. The time required to run the manual suites was somewhat comparable, but extremely variable depending upon operator response times.

6.1.5 Time to evaluate suites

QA test reports indicate that in the beginning, there is roughly a one-to-one correspondence between run times and the amount of time required to analyze the log files, although this diminishes to practically nothing as the end of the project nears.[22] Once again, this is variable. The logs indicate that one should allocate as much time to the analysis of the results as to the actual running of the suites. By running the proper combination of manual and automatic tests, it is possible to keep the test analyst busy running manual tests and then analyzing previous logs while the subsequent automated tests run. If one is fortunate enough to possess more than one implementation of the test harness, it should also be simple to verify the failures on the second harness while keeping the first harness "well fed" with test suites.

6.1.6 Time to correct suites

In almost all instances, the errors encountered in the test suites were simple misspellings of the operators. These can be corrected in a negligible amount of time, within the SRS itself.

Of particular historical significance was the effort involved in converting the test suites (at a very late date) to a "language independent" format. Language independence was required because the front panel messages could be specified in one of several languages: German, Japanese, English, Spanish, French, or Italian. Rather than duplicate each test for each language, we decided to revise the harness to understand front panel messages in any language. The conversion process involved changing the semantics of an operator which watched for front panel display messages. The operator was to be changed from a *postfix* oriented operator that used a character string parameter to a *prefix* oriented operator that used a constant.

[22] This is, of course, because the number of bugs will drop ... hopefully.

In this instance, the conversion was relatively easy to accomplish using a *PERL* script. A caveat however: when doing such an automated conversion, do not assume that the conversion will be complete. We chose a *new and different* name for the revised, prefix oriented operator and it proved to be a wise choice: the results were a *mostly* converted SRS, and the remaining unconverted operators proved to be a simple job for an editor.

6.2 Will we do it again? You bet! However....

Overall, the experience was a positive one. In order to accomplish the same result again, we must have some of the basic elements we had in the initial effort *as well as* some new concessions.

6.2.1 More realistic planning for implementation of test harness

"Now that we are done, we know how long it will take."

In the strongest sense of the word, this was certainly a high-risk approach in terms of *effort*: The technology itself was known and proven. The extent and success of this project relied entirely upon the QA engineers who tackled the problems. As is the case with unplanned projects, the amount of effort was entirely glossed over. As a result of doing this, we now have an idea of how long a modification of the test harness should take since we know what sections will have to change, and we can get an idea of their relative size by examining the current test harness.

In short, the next time around will be faster -- much, much faster.

6.2.2 Quality assurance group retains control of software requirement specification

In order to achieve any sort of success using this approach, the QA group *must* have control of the SRS. Without control of the actual framework for such a system, we are hopelessly lost from the start. Engineering specifications for testability are of the utmost importance to a quality product.

Quality assurance is the domain and responsibility of *every* engineer, but it is the charter of the QA group to ensure that it is achieved.

6.3 Acknowledgements and Disclaimers

I feel that every paper should contain appropriate acknowledgements and disclaimers.[23]

As the visionary of the *MOTHER* system, [24] I would be negligent if I did not give proper thanks to the people who actually made it work. The dirtiest and most horrific implementation problems were left to two outstanding engineers.

---

[23] Here are the disclaimers:

UNIX is a trademark of AT&T Bell Laboratories.

Turbo-C++ is a trademark of Borland International, Inc.

PC/AT and/or *Personal Computer AT* are probably trademarks of International Business Machines Corporation.

PC-NFS is a trademark of Sun Microsystems, Inc.

Ethernet is a trademark of Xerox Corporation.

VAX is a trademark of Digital Equipment Corp.

PERL, grep, pr, RCS, ms, sed, awk, Bourne Shell, troff, and just about everything else mentioned in this paper is a product that we *used* and did not develop: we are not claiming credits or rights to these things. All we did was glue things together in an interesting way.

[24] Someone once suggested that a visionary could be defined as "someone who is most likely hallucinating."

First, my most profound thanks to Doug Bingham, a fellow Software Engineer who specializes in design. Doug wrote at least *half* of the 1400+ requirements in SRS by himself, and a full 95% of the tests. He not only accomplished this in a prohibitive time frame, he *never* lost his composure when I said *"Just fabricate something, and we'll implement it in the test harness."*

Also, my heartfelt thanks to our hired gun, Alan Downing, who connected the test harness to the *Phaser III* printer and implemented a good number of the more arcane operators. Alan endured extremely long hours, changing priorities, and an absurd implementation schedule.

Thanks also to Gary Hanson for the *superb* suggestions for revisions to this paper. My thanks to Jan Maybee, Jack Slingerland, and Ross Taylor, who also proofread the final copy of this paper. (I hope I got everything right this time.)

Finally, my most profound thanks to our *electronics team* leader, Howard Goetz, and our project manager, Ron Adams, for their confidence.

Without the significant efforts of these people, you wouldn't be reading this. Guaranteed.

The T90 Project: Self-Restarting Parallel Automated Software Testing on Multiple Hypercube Architectures

Marc Baber, Walt Harrison, Gary Hartman, Debra Lee
{marc, walth, hartman, debral}@ssd.intel.com

Intel Supercomputer Systems Division
Software Evaluation Group
15201 NW Greenbrier Parkway, MS CO1-01
Beaverton, Oregon 97006

PHONE: (503) 629-7600

KEYWORDS:

Software Testing, Software Evaluation, Parallel Architecture, Hypercube, Crash Recovery, Fault Tolerant, Automated Testing, Parallel Evaluation, Configuration Management.

RELEVANCE:

The T90 project encompasses advances in automated software testing that include automatic crash/hang recovery and exploiting parallelism to speed up evaluation.

BIOGRAPHICAL SKETCHES:

Marc Baber received a B.S. in Computer Science from the University of Oregon in 1985 and is currently pursuing an M.S. at the Oregon Graduate Institute of Science and Technology. He has worked in the field of system software evaluation for high performance computer companies including Cray Research, Floating Point Systems and, currently, Intel Supercomputer Systems Division.

Walt Harrison received a B.S. in Aeronautical Engineering from Purdue University in 1971 and an M.S. in Sociology in 1974. He then moved to New York city where he worked as a computer analyst/consultant/instructor/programmer for research organizations, universities, banks and municipalities on a variety of projects. Since 1981 he has worked in Oregon as a system software evaluation engineer for Tektronix and Intel's Supercomputer Systems Division.

Gary Hartman has worked in the electronic systems business in a variety of careers including hardware technician, logistics management, technical writer, software development, and software evaluation. He currently manages the Software Production and Evaluation Department of the Intel Supercomputer Systems Division.

Debra Lee received a B.S. in Computer Applications Management from the University of Portland in 1984, and an MBA from U of P in 1989. She has seven years of experience in the field of software evaluation, and she has been employed by companies such as Floating Point Systems, Tektronix, and Test Systems Strategies, Inc. Debra is currently a Software Evaluation Engineer at Intel Supercomputer Systems Division.

The T90 Project: Self-Restarting Automated Software Testing on Multiple Hypercube Architectures

Marc Baber , Walt Harrison, Gary Hartman, Debra Lee
{marc, walth, hartman, debral}@ssd.intel.com
Intel Supercomputer Systems Division

Abstract

This paper describes experiences with an automated software testing environment in which multiple parallel architectures are tested simultaneously in a coordinated, fault-tolerant manner. The environment is built upon several widely-used Unix features, including a common working file system supported by NFS, architecture-specific executables and results supported by the make utility, and mapping of available tests to appropriate architectures/configurations via configuration environment variables used by the make utility. The problems of hanging tests and crashes of the system under test (SUT) are addressed by a system restart daemon (SRD) running on a relatively stable monitor system that detects hung tests and/or SUT operating system crashes, reboots the attached SUT, and restarts the automated testing procedure (skipping the test that was running last). A relational database is used to store test results from each version of the system software for comparison with other versions and for software problem reports.

A complete release evaluation can be turned around much faster because multiple host systems, with or without attached hypercubes, can participate in an evaluation ensemble, some acting as compile and link servers, while systems with attached hypercubes focus on execution of the parallel tests. Redundancy adds to the fault tolerance provided by the SRD to minimize the need for human intervention. With T90, it is now possible to leave a test suite running overnight on a half-dozen hypercube hosts. Each test driver might reboot its attached cube (SUT) perhaps five or more times with a particularly early version of a release or with numerous new tests that have not been run previously on all possible configurations. The combined benefits of the SRD and the ability to coordinate several systems in parallel allows us to perform testing in one weekend and two working days that would have required up to eight working days or more with our previous methods.

Introduction

Necessity was the mother of invention for the T90 project. We were in a position of having more tests than we had time to run and at the same time needed to write even more tests. One solution to this problem was to hire more people, but has some severe side-effects that we wanted to avoid. A better solution was to automate the way evaluation tests are run. The project began when, as a group, we asked ourselves, "How can we evaluate software better and faster by getting more results from existing resources? How should we perform *testing in the 90's?*". The project name, T90, is simply shorthand for the latter quest.

The problem of having more tests to run than time to run them is not a new one. The T90 solution gave us more time by parallelizing the running of tests. This basic approach will work with any computer system if there is a common interface between systems. The systems we test use a Network File System(NFS) [Sun86] interface as their common interface, and all are tied to a common NFS mount point. Thus, a common directory of tests feeds the systems. When we want more tests per hour, we simply add more computer systems to the mount point.

Instead of automating, we could have used people power. In other words, divide the work among more people and the more people you have the more tests you can run. This would require that each new person get a dedicated system of course. We could also have run multiple shifts. The reason we decided against the people power approach, besides cost, was motivation on the engineers' part to off-load the boring task of running tests. A typical release cycle will require all tests to be run on several versions of the software before it ships to the customer. Imagine sitting in front of a computer and running the same old set of tests once every two weeks for two months, and you will see the logic of using evaluation engineers to develop tests rather than actually run them.

It is necessary to recognize that very few things are free. There is some extra effort required to write an automated test for the T90 test driver . We are convinced, that the extra time spent is worthwhile. We also recognize that not everything can be automated, so one should also allocate time for freestyle bug-snooping.

The charter for our software evaluation group is to detect and describe defects in system software for the Intel line of hypercube multicomputers. The large number of possible configurations and the wide range of system software features coupled with the desires for automated testing and rapid turn-around have driven the evolution of a relatively sophisticated software testing environment.

This paper begins with an overview of the hardware and software that our software evaluation group regularly tests, along with historical problems encountered. Next, our design of solutions to the problems is outlined in more or less chronological order, following the evolution of the T90 environment. Following that, the success of the T90 project is discussed with regard to the project goals and the results achieved with the T90 environment during evaluation of a recent release. Finally, related work and ideas for future enhancements are presented.

Hardware Configurations

Our software evaluation duties require us to run tests on a multitude of hardware system configurations. We strive to run as many tests on as many hardware configurations as possible. However, time is our enemy and we pick the most likely configurations that are apt to be a problem. This scenario is similar to most companies in the business of offering a variety of systems depending upon the depth of the customers pocket-book. The scalable parallel supercomputer business amplifies this problem. In other words, a single test may be capable of running on tens or hundreds of typical single processor hardware systems while thousands or millions of configurations are possible for scalable parallel supercomputer systems.

The scalable parallel supercomputer systems business offers a new dimension in scalability since the system configurations offered are based upon the number of nodes in the system. For example, the iPSC/860™ is offered with anywhere from one to 128 nodes in powers of two. To complicate matters, a variety of node types are available. Each of these nodes are computer systems with options for available memory and attached co-processor.

Our system consists of a System Resource Manager (SRM) connected to a cabinet full of single-board computers called nodes. The exact number of nodes is some power of two from 0 to 7, meaning 1 to 128 nodes. Nodes communicate with each other by sending messages through a hypercube-topology network implemented on the backplane of the cabinet.[Int89] The SRM is also connected through a local area network (LAN) to remote host systems. A remote host system can be either a Sun 3, Sun 4, or Sun 386i workstation.

Generally, all nodes in the cabinet are of the same architecture, but there are four possible node architectures. A CX node is based on the Intel386™ microprocessor and has a Intel387™ floating point co-processor. An SX node has a Weitek floating point accelerator instead of the standard Intel387™ co-processor. A VX node is a double-board computer that includes a CX node coupled with a vector floating point unit. Finally, an RX node is based on the Intel860™ microprocessor. All four architectures can run many of the same tests, however different compilers and/or options are necessary to produce the appropriate test executables.

Optionally, a hypercube may have one or more I/O nodes which are similar to the CX node except that they have only one communication channel which connects them to an anchor node within the hypercube. A SCSI bus interface is included on each I/O node to accommodate I/O devices which may include hard disks, 8mm tape or 9-track tape units. The I/O node can also provide a connection to a local area network

Bottom line, we have a lot of combinations of configurations that need to be tested. An automated solution to start a test, track its results, and in general "baby-sit" the test execution makes sense.

Subject Software

The iPSC software consists of:

Operating systems (for the SRM and nodes)
Networking software
iPSC extensions (commands, libraries, and server processes)
Diagnostic programs

The SRM runs the UNIX System V operating system with iPSC extensions that interface to the nodes. These extensions let you allocate and deallocate cubes, load programs on the nodes, and obtain node status information.

Each node runs the NX/2 operating system, which provides message-passing capability, memory management, and process management. The NX/2 operating system also manages the numeric co-processor and optional vector processor.

The SRM and the customer's workstations (called remote workstations) also run the TCP/IP networking software. This software lets you remotely log into other systems on the network and transfer files between systems.

Optionally, the SRM and remote workstations can run the NFS network file system software. NFS allows files to be shared transparently over a local network. NFS lets you access files on other workstations as though they were resident on your own workstation, thus eliminating the need to log into another system and copy the files.

Another system option is the CIO Ethernet option. This consists of TCP/IP running on the RX nodes of the iPSC system, and X Window client libraries that allow you to run and develop X Window applications that run on the RX nodes.

The iPSC extensions contain application libraries, shell commands, and several background server processes (or daemons). The libraries provide iPSC system calls that are available to both host and node programs. Several daemons enable message passing between the nodes and the host. A typical background process is the file server (called fserver) that runs on both the SRM and remote hosts. The fserver process allows the node to run standard I/O functions on the host file system.

An iPSC application can have a host program that runs on either the SRM or a remote host, and one or more node programs that run on a group of allocated nodes called a cube. iPSC commands let you control the allocation and operation of a cube.

In summary, the software tested includes NX/2, a UNIX-like operating system for the nodes, a concurrent file system (CFS), an interactive parallel debugger, TCP/IP network support, X-window client routines, a profiler, support for cube access by remote host programs, C and Fortran compilers and other specialized tools and libraries[Int90]. In other words, there is a lot of software much of which is optional.

The added dimension of optional software adds another twist to the need for more environments in need of testing. The problem of more environments needing testing and less time available to actually run tests is typical of most of today's computer systems sold in the commercial market. The T90 model of automation, providing unassisted testing across a wide variety of platforms can be adapted for use in almost any testing environment to achieve the same benefits we have realized.

Historical Evaluation Problems

Prior to the T90 project, testing was performed by a few evaluators each with their own hypercube system configured specifically for the system software they had to test. Each evaluator proceeded independently and was generally able to cover one or two architectures for their subset of the system software within one evaluation cycle.

Automation was provided by shell scripts and the make utility, but since the default rules of make provided no distinction between objects and executables of different node architectures, one architecture had to be completed before another could be begun. In addition, test builds and runs were typically separated into two tasks. Therefore, at least four to eight evaluator interventions per system were required to test all four architectures for each major software feature. In practice, hanging tests and hypercube crashes caused by preliminary versions of a software release resulted in many more interventions and frequent delays since the anomalies tended to occur during overnight runs when evaluators were not usually available.

The problems with the old approach were several. First, evaluators spent much of their time running tests and had little time for fault analysis and new test development. Secondly, the hardware was poorly utilized because of problems with overnight test runs and because it was next to impossible to schedule rotations of hardware between evaluators to cover all the hardware configurations. Finally, the proliferation of both new hardware and new software products overwhelmed the evaluation group, leading to low software coverage.

Goals for the T90 Project

To address these problems, the T90 project was initiated to modernize our software evaluation methodology. The specific goals of the project were:

1. To be able to use any available SRM on the internal ethernet which has the current system software installed.

2. To utilize available hardware and people more efficiently.

3. To run tests in a shared cube environment in order to test the emerging customer environment as well as meet goal two.

4. To make automated regression testing proceed in a fault-tolerant manner so that an evaluator could reasonably expect a test suite to run all night or all weekend once the test driver was started.

5. To provide automatic hang detection, cube rebooting, and test driver restarting performed by a daemon with an absolute minimum of repeated work after a restart.

6. To design tests to produce result files that can automatically be stored in a relational database for automatic tracking of bug status.

7. To enhance ease of use for evaluators, technicians, contractors and developers.

8. To combine the test compilation and execution steps into one step.

9. To facilitate the simultaneous testing of all architectures in the same test directory.

10. To allow tests to be run in any order, ultimately using the data base result records to schedule the most productive tests first, or for overnight stability, to save the tests most likely to hang the system for last.

A Flexible Test Driver for Parallel Testing

The first task of the T90 project was to centralize the evaluation test code and execution scripts onto a single system, to simplify source management and provide for easy backup. Access to this central test base by multiple systems is easily provided via NFS mounts.

With a central test base a test driver can be executed on any system, to execute the tests for the hypercube architecture attached to that system. The test driver needs to know how to traverse the central test base directory structure, finding all directories with tests to be executed. To support multiple test drivers working against a central test base, a locking mechanism is required so that no two test drivers attempt to generate results in the same test directory simultaneously. The test driver should manage an evaluation pass, making one complete pass, or circuit, of the central test base attempting to generate results before deciding that testing should stop. If a test directory was locked on one circuit, then additional circuits (including only the remaining directories) should be attempted. The test driver needs to support a set of environment variables [Bab90] for the given system, and to execute the make utility [ATT89] with a master makefile. With these easily coded C shell functions the test driver is simple and flexible.

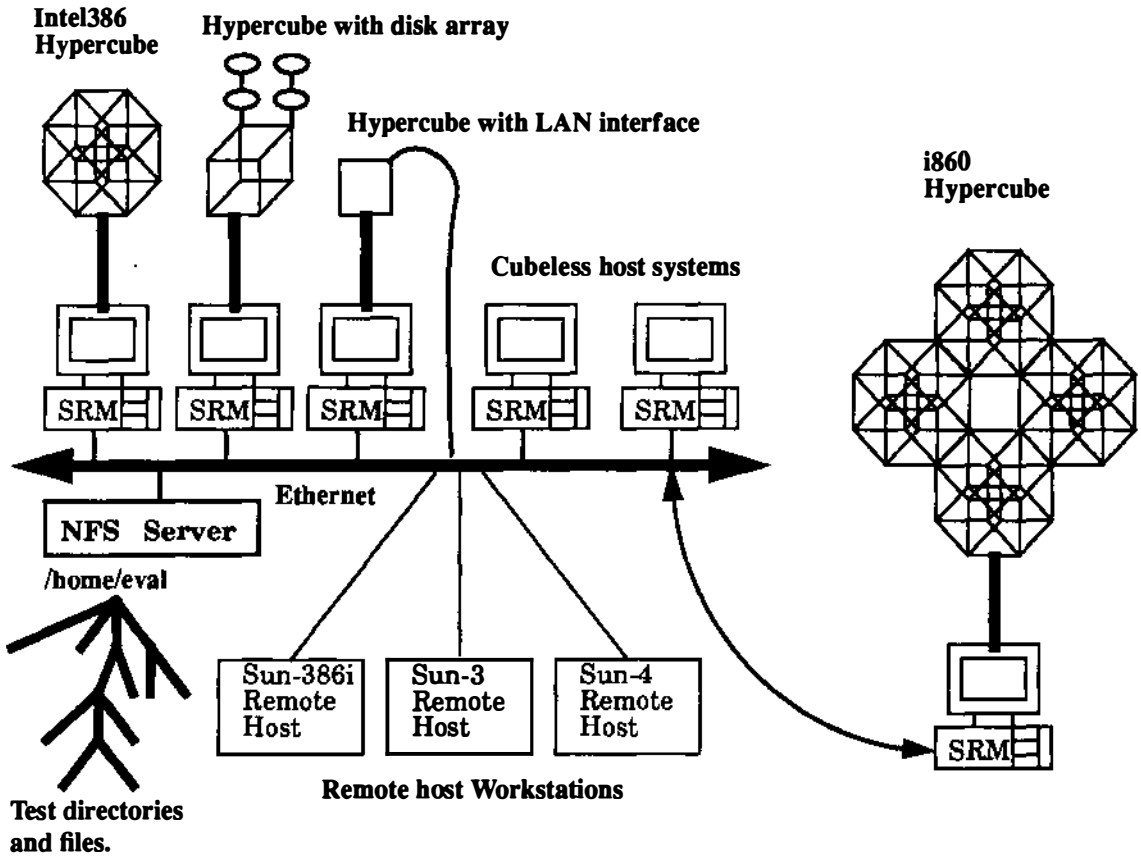


Figure 1: Testing within the T90 environment. Multiple host systems (SRMs) with various hypercube configurations each execute subsets of the tests stored on the NFS server, depending on their hypercube's hardware configuration. Workstations running remote host software run the same tests in the remote host mode. Cubeless systems act as compile servers.

Figure 1 shows an overview of a complete ensemble of several hypercube configurations with both local SRMs and remote workstations executing the test driver in parallel. Environment variables define the size and type of hypercube architecture attached to the system, and are used in the master makefile. The master makefile defines separate object and executable suffixes for each architecture and the rules to compile the test code as well as the rules to execute the test code and log the results.

A typical T90 test directory contains one local makefile, one Bourne shell script to execute the test and log results, and any test code for the host and/or node. The test code, currently C or Fortran, usually contains the actual test to be evaluated, with a PASS/FAIL output.

The test driver is designed to process each test directory until it successfully runs the make utility in every test directory. The make utility is invoked to use the environment variables and read both the master makefile and the local makefile for the test. If a test directory is locked, by another system, that directory is placed on the list of directories for the next iteration, and processing continues with the next directory in the current list. Also, if the make utility terminates with an error exit that test directory is added to the next iteration list. The test driver also supports various command line options to specify alternate makefiles, test directory lists and/or number of iterations to list a few examples.

Using "make" to Build and Run

In order to combine the test compilation and execution phases of regression testing (goal eight), it was decided to add test execution rules and test result targets to makefiles that were already used in the compilation phase. This was easier and more flexible than adding test building commands to the test run scripts. The make utility also provided an added benefit in that tests are not rerun if their result files are up to date, so when make executes in a partially completed test directory, it does not repeat any work done by previous test drivers. This feature, combined with the test driver's ability to resume testing in the next directory after a restart met goal five for avoiding unnecessary repetition of work.

The test driver and the SRD know absolutely nothing about test procedures, they merely facilitate running a large number of tests in a reliable manner. All of the information about building and running a test is contained in its local makefile, the master makefile shared by all tests and the test commands (scripts or programs) themselves. A test author can therefore write tests that perform practically any test procedure that can be invoked by a shell command.

By carefully defining suffix rules in the master makefile, it was possible to minimize the size of the local makefile needed in each test directory and to rely on the master makefile for rules to build almost all targets. In addition to rules for objects and executables for each of the architectures, there are rules for producing execution log files (running the tests), and result files.

Result files contain a single line, each summarizing their corresponding log file including the name of the test, the hypercube hardware configuration used (encoded in the result file's name), the network name of the host system, a pass or fail result, and how many seconds the test took to execute. The execution time is to be used for weighting test results, discovering performance degradations, and estimating MTBF (Mean Time Before Failure) for the system software.

Included in each test directory's local makefile are lists of sources, objects, executables, log files and result files to be generated plus any build and run instructions that are not covered in the master makefile. Both the local makefile and the master makefile are necessary to run a test (unless no default rules are used, which is quite rare).

The master makefile contains generic instructions for building and running tests based on a set of file suffixes that distinguish all known file types in the iPSC system. Figures 2a and 2b show the relationships between the different file types/suffixes. Each arrow in the diagram corresponds to a suffix rule in the master makefile.

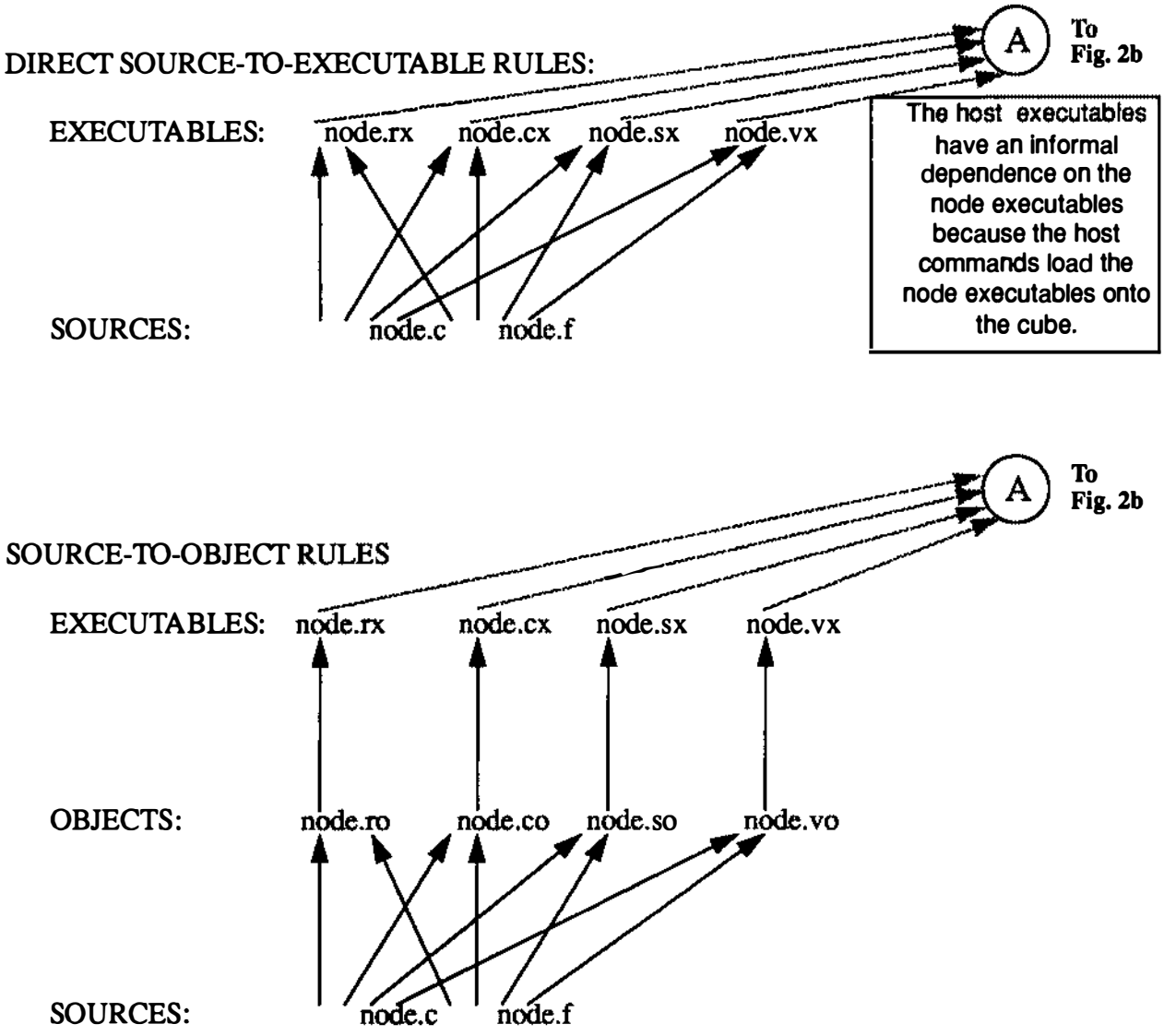


Figure 2a: Rules for Node Executables. All tests must have a host process of some kind, so there are no rules to generate log files or result files from node executables. The host executable is the only place where the knowledge about what node executables to run is stored, so the host and node programs may have different base names without limiting the utility of the default suffix rules, as long as the host executable uses the correct basename for the executable and appends its own architecture argument as the suffix.

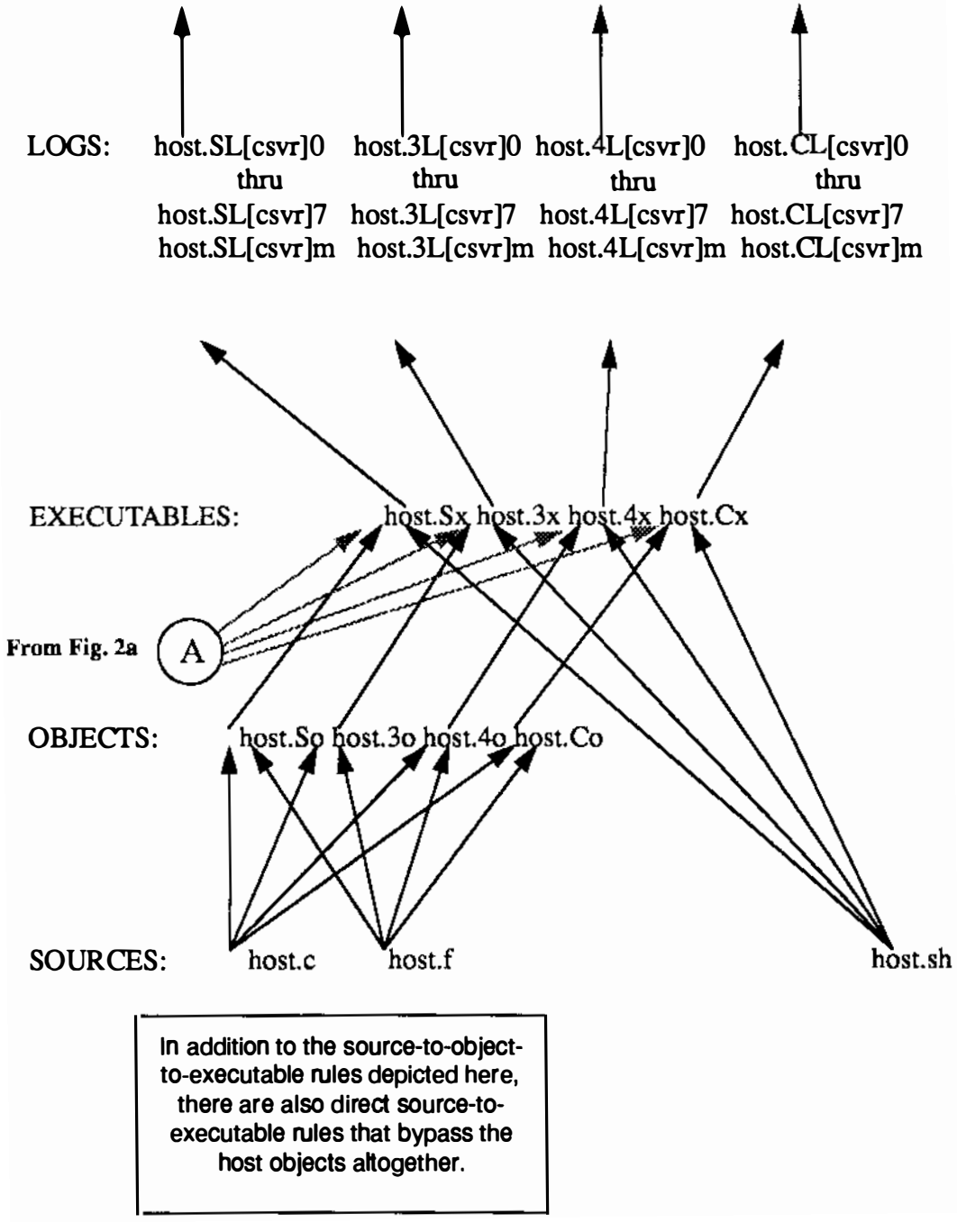


Figure 2b: Rules for Host executables and Log/Result Files. Log and Result files are derived from host executables. The informal dependence upon node executables is enforced by deleting all log and result files in a directory whenever a new node executable for the same architecture is created.

The suffixes are defined as follows:

.c .f	C and Fortran source files
.co .so .vo .ro	Node object files (CX, SX, VX, RX nodes respectively)
.cx .sx. .vx .rx	Node executables (CX, SX, VX, RX nodes respectively)
.So .3o .4o .Co	Host object files (SRM, Sun-3, Sun-4, Sun-386i hosts)
.Sx .3x .4x .Cx	Host executable files (SRM, Sun-3, Sun-4, Sun-386i hosts)
.[S34C]L[csvr][0-7]	Log file from runs, for example: .SLv5 indicates a log file of a test run with an SRM host and 32 VX nodes. First character indicates host type, "L" indicates log file, third character indicates node type and fourth character indicates dimension of the hypercube.
.[S34C]R[csvr][0-7]	Result files, everything but "R" has the same significance as for log files.

There are also t90 suffix rules that allow one to request a test be run with the largest available cube of a specific type. These suffixes are the same as the log and result suffixes except that the cube dimension is replaced with "m" for "maximum." Whereas the .SLr0 thru .SLr7 rules would request specific sizes (d0-d7 on an SRM host), the .SLrm rule requests the largest available RX cube. When multiple SRMs are participating in an evaluation, the first test driver that enters the directory and is configured with at least one RX node generates a result file with its largest RX subcube. Subsequent test drivers on other SRMs will decline to rerun the test if their largest RX subcube is no larger than the existing results subcube, but will run the test and overwrite the result file if they have access to a larger RX configuration.

The master makefile has grown to over 1700 lines at present, more than 1500 lines excluding comments. Though the file does handle every combination of host architecture, node architecture and cube size, its size has become a bottleneck because it takes the make utility over two minutes to parse all of the rules and the parsing is repeated in every test directory. Future versions of the test driver will make use of the hierarchical features of make to avoid multiple parsing. Another prospect for improving the situation is to use only the appropriate subsets of the master makefile for each driver. For example, if a driver is running on an SRM it does not need access to rules for the other hosts.

Managing Configuration Constraints

Because the software loaded on a system is independent of the hypercube architectures, it is possible to harness many systems to compile the test code sources. This is typically a one-time-only task per software transmittal, and requires that test executables be generated for each hypercube architecture. Since this compilation task is early in the evaluation of a software release many systems and test drivers can be executed in parallel, and the hypercube hardware need not be present. The environment variables [Bab90] contain the size and type of hypercube architecture and are used in the master makefile to determine which test results should be generated by comparing the requested hardware (encoded in the files suffix, as in Fig. 2a) with the available hardware, described by environment variables. If no hypercube architecture is present the test sources can still be compiled.

Individual test drivers take responsibility for attempting to generate test results only for the hardware present, using the environment variables. The host system, node architecture and hypercube size are all encoded in the suffix for the test logs and results generated. This allows make to use the test result suffixes to determine which test results to attempt, given the node architecture available on any given system, see Fig 2b. Since the master makefile contains both compilation and execution rules, both steps can be performed for any given test directory, provided the configuration supports the results to be attempted. For hypercubes with I/O sub-systems the configurations are too numerous to encode in the file suffix, so the individual tests must manage these environmental variables and exit gracefully if the hardware will not support the test results. A graceful exit implies zero, so that the make utility will assume a successful test and the test driver not attempt the test again on its next iteration. For systems without appropriate hardware the master makefile rules merely print a notice to that effect, and exit gracefully. This does not inhibit another test driver on another system from attempting to run the test if the hardware will support the results.

One mechanism used to help manage which tests are attempted is to limit the test driver to a specified test sub-directory, i.e. Fortran, so that only those results will be attempted. This can be done as long as the central test base is structured in some formal and logical way. The central test base was structured to generally match the hypercube system documentation as shipped to customers. The secondary sub- directories are for the software class, C or Fortran; and the tertiary sub- directories represent the level of software control, global, message, cube ... By limiting the test driver to a single sub-directory we were able to control which tests were attempted.

Another technique used to help manage which tests are attempted is to manually alter the environment variables, to restrict the size of the hypercube available on the system or to mask certain hypercube architectures entirely. Of course, by explicitly listing directories to be executed in a file, one can direct the test driver to attempt any subset of the test directories.

A System Restart Daemon

To meet goal five of the T90 project, a System Restart Daemon (SRD) was implemented that monitors the progress of each test driver running on its host system. If any test driver makes no progress for over twenty minutes, all test drivers are killed, the local hypercube is rebooted, and all the test drivers are restarted. The test drivers quickly proceed to the points where they left off because the list of remaining directories is kept up to date in the /tmp directory or each test driver and because make does not regenerate files that are up to date. The test directory that was suspected of causing the system crash or test hang is left locked for later inspection so that the test driver does not get stuck on a single hanging test.

Communication between the SRD and test drivers is achieved with the time stamps of "pulse files" in the /tmp directory.

The master makefile contains progress reporting in several key rules so that most tests do not need to worry about reporting progress. Tests that normally run over twenty minutes must update the time stamp of a "pulse file" periodically to avoid being aborted. The time limit of twenty minutes was selected because very few of the tests in the test base run longer than that and it is short enough to detect tests that are hung.

The mechanism for detecting hanging tests is simple and it is based on standard Unix commands. The daemon is written in C-shell except for several routines written in C for detecting anomalies other than hanging tests. The SRD executes as a background command running in an infinite loop. Every twenty minutes, the SRD "wakes up" and performs an "**ls -lt**" of all the test

driver's pulse files and its own pulse file which retains the time stamp from the previous iteration. If all of the test drivers have progressed in the last twenty minutes then all of their pulse files will have been touched (with the Unix **touch** command) at some point in the past twenty minutes and the SRD's own pulse file will be the last file in the list. The output of the **ls** command is piped into the unix **tail** command to test this assertion. If all is well and the assertion is true then the SRD **touches** its own pulse file and goes back to **sleep** for another twenty minutes. Notice that the time required to detect a hanging test with this method is between twenty and forty minutes, the average time being thirty minutes.

If the SRD determines that a test driver has hung, it uses the **kill** command to kill each test driver process. The **ps** command is used to detect orphaned processes which are also **killed**. After all the test driver processes and their descendents have been **killed**, the SRD causes the hypercube system to be rebooted. After the hypercube is rebooted, the individual test drivers are all restarted using the commands recorded in their respective pulse files. The SRD defines environment variables that inform each test driver what directory to resume testing in based on the last directory noted in each of the test drivers' log files.

Unfortunately, detecting lack of progress is not enough. A useful SRD must also restart when erroneous progress is being made. Therefore, whenever a test gets an unexpected failing result it causes make to execute a quick system confidence test to detect if the system software has become corrupted. If so, the test driver creates a file that requests a system restart and shuts itself down. At the next invocation of the SRD, the request file will force a system restart. Examples of a corrupt system include a system that cannot allocate any subcubes, a system whose disk is full, or a system whose communication servers have aborted. The list of conditions to check for is relatively short, but it can be expanded in the future to accommodate whatever conditions become limits to testing productivity. Figure 3 shows the decision path for the make utility which allows it to detect erroneous progress and flag the SRD, if necessary. The flowchart is encoded as a single 57-line rule in the master makefile that controls the generation of all result files.

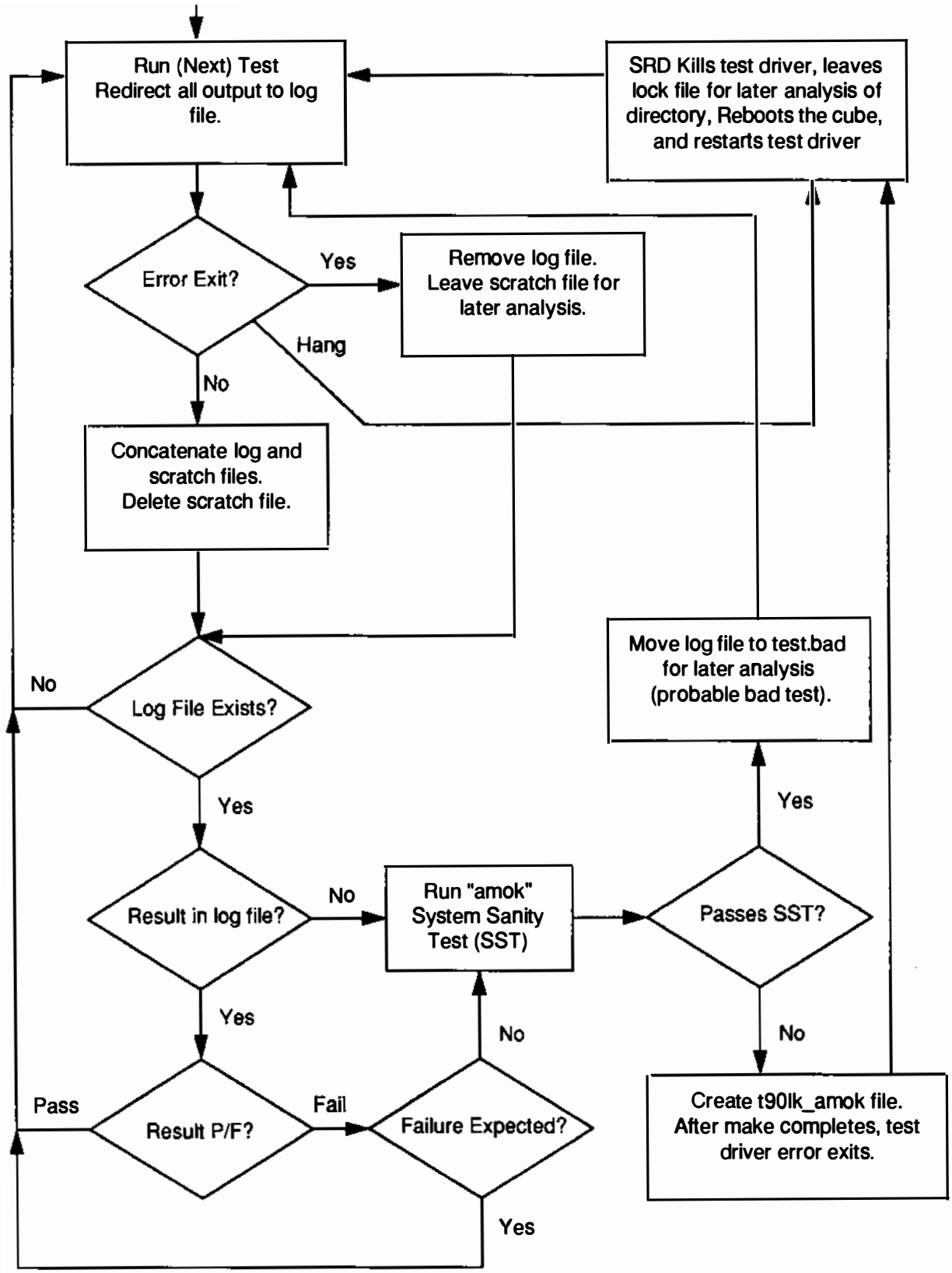


Figure 3: Flow of Control between Test Driver, Amok, and the SRD

Figure 4 shows all the interactions between processes necessary to run the T90 test driver on a host system. Although normally a user only directly invokes the SRD and the test driver, other processes, invoked by the test driver, are also necessary. The **hostSys** and **cubeSum** commands provide information to the test driver that allows it to define a set of configuration environment variables. The environment variables are inherited by the **make** utility when it is invoked by the test driver. When the status of the attached hypercube is in doubt, the rule that generates result files causes **make** to invoke the **amok** command which performs a quick test of the hypercube. If **amok** finds a fault, it returns a non-zero exit status to **make**, causing **make** to return an non-zero exit status and create a lock file that signals the test driver to terminate. The **amok** command also creates a file (/tmp/t90_rebootcube) which signals the SRD to force a system restart if the hypercube is determined to be in an unusable state.

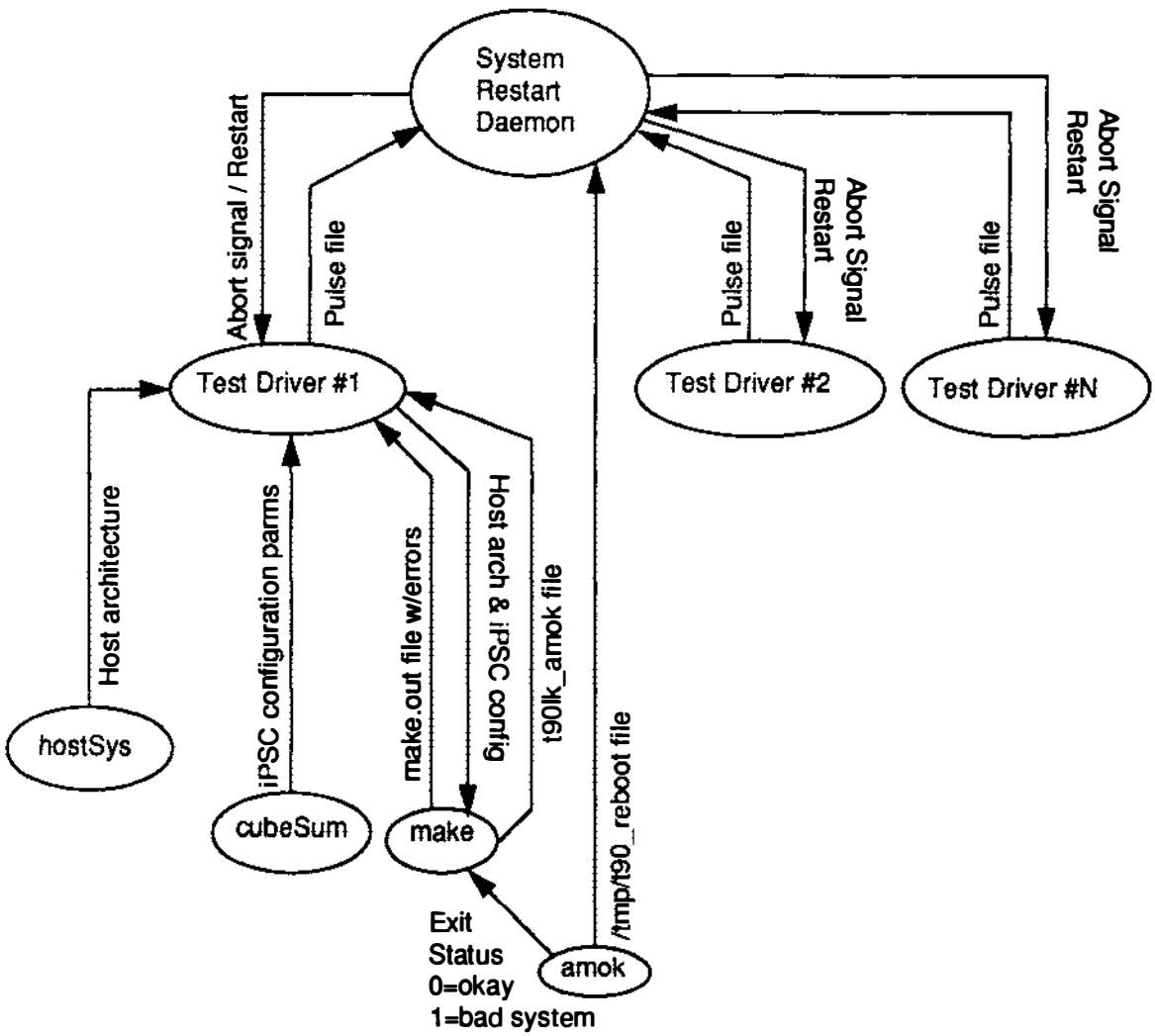


Figure 4: T90 test automation executables on each SRM host (Data Flow Diagram)

Tracking Results with a Relational Database

T90 is also coordinated with a problem tracking system, which is used when bugs are submitted. This allows bugs to be matched up with a corresponding T90 test for which that bug can be verified as fixed, or for which the t90 test can verify that a bug exists.

Once all the test results are analyzed, an **awk** script condenses all the information from all the result files into a file of SQL (Standard Query Language, or "sequel") statements. These statements are then used to update the results database [Ora88]. Information contained in the SQL statements are test pathname, the configuration and name of the hardware system the test ran on, and whether the test passed or failed. With this information, the database can quickly locate any failing results for which no bug reports exist as well as unresolved bug reports whose test cases are no longer failing. The goal is to ensure that every bug report has a T90 test case. When a test case is linked with a bug, it gives the software designers a reference to begin fixing the bug, and it gives the evaluators a way to verify that old bugs have not resurfaced.

Project Results

The single most outstanding result of T90 was that two additional software transmittals were required in the software release which resulted in a better software release. This achievement was the result of the number of test failures the evaluation team was able to generate in the early release cycles; which forced the two additional software transmittals into the overall software release schedule.

As no surprise, there was additional debug time required as the test driver was implemented; but this also produced some surprising results. Initially, as the test driver was implemented T90 began generating more results than the evaluation team could summarize. In fact, the number of initial test failures took an unscheduled amount of time to analyze, delaying one software transmittal.

Almost five hundred tests now conform to the standards required by the test driver. Each test can produce one or more results for each configuration, which achieved project goals #8 and #9. At last count, T90 generated over 1700 results during a recent evaluation. The SRD is reliable enough to keep test drivers running on several hypercube systems for a full weekend without human intervention, satisfying project goals #1 and #2. In one instance, three systems each required five or six hypercube reboots over the course of a sixty-two hour weekend, and all three test drivers were still running and generating useful results on Monday morning, project goal #4.

After most of the tests have been compiled typical figures show that 250 to 550 results can be generated in one 24-hour period. This variation depends on the number of systems participating, how much compilation is required and how many results need to be generated. Generally, more systems generate more results, but also increases contention for access to central test directories. Early in the evaluation process more time is spent compiling the test sources and tends to reduce the number of results generated. And, later in the process there are usually less results to generate and more time is spent by the test driver in traversing the central test directories.

Another outstanding result of T90 was the SRD, achieveing project goal #5. Without the ability to monitor, stop and re-start an automated test driver the productivity gained would be limited to the first test that hung the hypercube. With the SRD we can be assured that the test driver is still attempting tests around-the-clock and without human intervention.

Future Research

T90 is also a system that can accept test cases created by others outside of the software evaluation group. The tests will need a wrapper that will fit with the test driver. This creates a more thorough test base and contains tests that the software evaluation group would not have the time or expertise to create. For instance, the X Testing Consortium Xlib tests have been slightly altered to fit into the T90 system, and that will provide approximately 350 tests to be added to the test base, with very little actual work done by the software evaluation team. Also, there are plans to include test cases created by the Applications group. T90 will then contain some real world examples of programs running on the Intel Supercomputers.

The database format has been set up to contain the size of each software component in KLOC (1K lines of code) as well as the execution time of each test. We plan to use the information to produce metrics of test coverage such as test execution time per KLOC of system software code or number of failures observed per unit of time per KLOC of system software code, and so on. The database will allow us to experiment with many different metrics and to choose which are the most useful as more experience with the system is gained.

Another goal of T90 is the idea of a "push-button transmittal" which would allow automatic through-the-network updating of system software being tested. This could further improve evaluation turn around time.

The system restart feature depends upon having a stable system connected to the system being tested. At first, this may appear as merely pushing the problem to a new location (what if the host system crashes?). We contend that current networking technology makes it possible to extend the restart feature to heterogeneous systems, each capable of monitoring and restarting the others. The more systems involved, the lower the likelihood of simultaneous failure.

Short-term research for T90 will be to decrease the time spent in any given test directory and to decrease the time spend traversing the central test base.

The first goal will investigate customizing the master makefile, to reduce its size for any given invocation of the test driver. This may mean constructing a makefile for each test driver invocation as a function of the software available and the node architecture.

The second goal will investigate how to provide the test driver a list of tests to be attempted based on results needed, not by simply traversing the central test base looking for results to generate. This could be provided by the relational database described earlier.

Longer term research for T90 would convert the test driver script to Bourne shell to take advantage of functions and multiple signal handling capabilities. Some of the SRD functions could then be moved into the test driver, creating a more robust test driver as well as reducing the test driver/SRD overhead.

Related Work

In Japan, testing tools are ranked as the number one necessity for future productivity improvement [Nom87]. However, tools are not being used there and the level of people's knowledge of them is still rather limited. Thus, testing becomes the most troublesome phase in their software development process. Furthermore, since turnover rates of engineers in Japan is much lower compared to the United States, the usage of testing tools in the maintenance phase is lower than it is in the U.S. If there are problems with the software, the original author is still available for consultation.

Consequently, the need for testing tools during the maintenance phase may be seen as unnecessary because the confidence of the bug fix is higher if the original designer is modifying the software. This is contrary to the environment here is the United States, where there is a higher employee turnover rate, and often a sustaining engineering team is assigned to the bug fixes. The sustaining team usually is not the original authors of the software.

Henderson at Hewlett-Packard described an automated test controller called Big Brother [Hen89], consisting of two tools, Suiterun and MSPSRUN. Suiterun initiates one or more MSPSRUN commands and provides a method of specifying groups of tests to be run in parallel. Each MSPSRUN command sequentially executes test run scripts from a collection of tests in one "volume". In contrast, parallelism in running T90 tests is implicit and occurs when multiple test drivers are executed. In order to test in parallel on a single system, multiple test drivers can be initiated on a single host. Midway through the Big Brother project, it was observed that one of the remaining challenges of the project was that "Human intervention to detect system failures and perform dumps was required." As a solution, a tool called "Multitree" was developed to detect system hangs and even performed memory dumps unattended. The capabilities of the Big Brother system included the possibility of multiple actions to correct a failed SUT (System Under Test) including the ability to place a phone call to an evaluator and inform him or her that an irrecoverable error had occurred, essentially a call for help made possible by a voice synthesizer. The T90 SRD has not reached this level of sophistication, partly because there is no cost justification for adding more heuristics if we are able to keep our test suites running with 90-95% certainty that no catastrophic errors will occur during our absence.

Conclusions

The T90 project has achieved many of its goals. The T90 test system exists in one location, and any SRM that has the current software installed can access the directories via NFS. This facilitates simultaneous testing of the software in the same test directory. T90 can also detect if tests for the desired hardware have already been run and T90 will not run them again, and since T90 has environment variables that can be changed to fit the size and type of hypercube architecture that the user desires, tests can be run in a shared cube environment. These two features allow the hardware to be utilized more efficiently. The SRD provides automatic hang detection, cube rebooting, and test driver starting. With the SRD, the evaluation team can start a regression run over night or during the weekend and it would continue to run without any human intervention. The T90 results have also been stored in a relational database for future reference. This helps the evaluation team, based on past history, to schedule the most productive tests first and save the most troublesome tests for last. Storing the results in a database also matches test suites to reported bugs to aid in tracking the bugs status. The main users of T90 have been the evaluation team; thus, it has seen little use by technicians, contractors, and software developers. Even with the first use of T90 on a project, the general consensus throughout the company is that T90 has provided great benefit, not only with test execution, but also with the ability to connect with the bug tracking system. This, in turn, will lead to improved software quality in the Intel line of supercomputers.

References

[Sun86] Sun Microsystems, Inc., *System Administration for the Sun Workstation*, Publication 800-1323-03, Revision B of 17 February 1986, Sun Microsystems, Inc., Mountain View, Calif., 1986.

[ATT89] AT&T, *UNIX System V/386 Release 3.2 Programmer's Guide*, Vol. 2, Chapter 13, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[Ora88] Oracle Corporation, *SQL Language Reference Manual Version 6.0*, Publication 778-V6.0, Oracle Corporation, Calif., 1988, 1990.

[Int89] Intel Scientific Computers, *iPSC/2 System Administrator's Guide*, Chapter 1, Publication 311014-004, Intel Scientific Computers, Beaverton, Ore., 1989.

[Int90] Intel Scientific Computers, *iPSC/2 iPSC/860 User's Guide*, Chapter 1, Publication 311532-006, Intel Scientific Computers, Beaverton, Ore., 1990.

[Bab90] Baber, M., *T90 Phase 3: Test Automation Specification*, Internal Specification, Intel Supercomputer Systems Division, Beaverton, Ore., 1990.

[Nom87] Nomura, Toshitsugu, Use of Software Engineering Tools In Japan, ACM, 1987.

[Hen89] Henderson, B. M., "Big brother --Automated Test Controller." Sixth International Conference on Testing Computer Software, Washington, D.C., May 22-25, 1989. 455

Predicting Error-Prone Modules in a Large Evolutionary Development Environment

by

James S. Collofello
Eric Wagner
Computer Science Department
Arizona State University
Tempe, Arizona 85287-5406
(602)965-3190

Abstract

The process of producing a new release of a large software system is a time-consuming and error-prone process. This process can be improved by restructuring the software system and improving defect detection processes. Unfortunately with the limited resources available to most projects it is impossible to apply these improvements to all modules. Thus, a cost-effective compromise is to identify those modules which are more error-prone than others in the system and apply the improved defect detection processes to them. This paper describes our research approach and initial results in attempting to identify error-prone modules in a large evolutionary development environment.

Biographical Sketch

James Collofello received his Ph.D. degree in Computer Science from Northwestern University in 1978. He is currently an Associate Professor of Computer Science. His research interests include software engineering, software quality assurance and software maintenance. He has researched and consulted extensively over the last ten years in the software engineering and software quality assurance areas with several large companies. He has also written many research articles in this area.

Eric Wagner received his B.S.E. degree in Computer Science from Arizona State University in 1979, and his M.S. degree in Computer Software Engineering from Arizona State University in 1989. Presently he is the Vice President of Product Development for Ithaca Software, developer of the HOOPS 3D Graphics System. Prior to working at Ithaca Software, he was the Manager of Quality Assurance and Quality Control at AG Communications Systems.

Background

Much attention has been focussed recently on predicting error-prone modules in a software system. Error-prone modules are those modules which have a higher probability of failure than other modules in the software. If these error-prone modules can be identified, then they might become the target of restructuring activities or increased defect detection processes. Unfortunately, much of the research in the area has concentrated on identifying error prone modules in new development projects. This paper attempts to address this shortcoming by assessing error prone module prediction approaches applicable in large evolutionary development environments. In addition, it seems that newer and more-complex predictors are being invented everyday. This paper compares results of some of the newer, more complex predictors against some of the more basic, traditional ones.

In the context of this paper, a large evolutionary development environment is defined as one which meets the following criteria:

- the software product is extremely large and complex (100,000 lines of source or more), subdivided into many subproducts (modules), with numerous interfaces existing between the many software subproducts,
- many designers and maintainers participate in the development (100 or more), and
- the product development interval is long (1 year or longer).

These criteria lead to the following problems that are more prevalent on larger projects than smaller ones:

- Very large and complex software products will challenge maintainers in their abilities to understand and comprehend the intricacies of the operation and interfaces between the numerous modules (high psychological complexity). This can make it very difficult to locate all areas affected by a maintenance change - the probability of large "ripple effects" is quite high.
- Having a large number of designers and maintainers, each working on one, small portion of the overall system, makes it difficult for any one person to become a "system expert," knowledgeable in the function of the system as a "system." Instead the information is fragmented over dozens (or hundreds) of people.
- Long development intervals are inevitably accompanied by large amounts of personnel changes. Many of the designers and maintainers will be rotated into other design positions, other management positions, or will leave the company altogether. Also new employees will be added to the project on a continual basis to account for attrition. This can make it difficult or impossible for a maintenance programmer to locate or access the original developer, which is a severe handicap during maintenance.

Thus, in a large evolutionary development environment the ability to predict error-prone modules involves identifying those modules which have a high probability of being modified incorrectly, i.e., they possess poor maintainability attributes.

In the remainder of this paper, our research approach to developing a set of practical error-prone module predictive measures for large evolutionary development environments will be described. Some initial results from a case study will also be presented.

Research Approach

Our approach to identifying error-prone modules involves 3 sets of measures as defined below:

1. code-based measures: measurements which are based on access to the source code
2. documentation-based measures: measurements which are based on the module's documentation and not its code
3. history-based measures: measurements which are based on historical data collected about the module

Examples of code-based measures include:

1. lines of code
2. McCabe measures [1]
3. Halstead measures [2]

Examples of documentation-based measures include:

1. number of functions performed by a module [3]
2. external documentation quality [4]

Examples of history-based measures include:

1. software age [5]
2. number of revisions
3. previous defect history

It is our belief that the most accurate prediction of error-prone modules will require analysis of information from each of the metric sets. Thus, our goal is to investigate the performance of various candidate metrics from each of the sets and eventually to pinpoint the best combination of metrics from each set. This paper discusses our initial results in assessing a candidate set of code-based measures. Our set consisted of well-known metrics which are practical to collect in a large evolutionary development environment. The metrics were chosen based on their ease of calculation with the belief that simple metrics should be evaluated before more costly metrics are considered. The metrics are identified and briefly described below:

1. NCSS - The number of non-commented source statements.
2. SS - The total number of source statements.

3. CPLX - A McCabe-style complexity metric consisting of the sum of the number of "IF," "CASE," "WHILE," "REPEAT" and "FOR" statements added to the number of procedure calls made by the module.
4. LINES - The number of lines contained in the module.
5. COMM - Lines of comments (LINES - NCSS).
6. COMMD - Comment density (COMM/LINES)
7. MCCB - A McCabe-style complexity metric consisting of CPLX - (number of procedure calls made by the module).
8. MCCBD - McCabe density (MCCB/SS).
9. ENTE - The number of externally-visible entry points to the module.
10. ENTT - The total number of procedures and functions declared within the module.
11. PARM - The number of parameters passed into and out of all procedures and functions declared within the module.

Case Study

In order to assess the ability of our candidate set of code-based metrics to predict error-prone modules a case study was performed. Thirty modules representing a cross section of a 1,200 module system undergoing evolutionary development were selected for application of the metrics.

NCSS	SS	CPLX	LINES	COMM	COMMD	MCCB	MCCBD	ENTE	ENTT	PARM	ERRORS
2743	1092	210	5550	2807	.5057	158	.1446	1	2	47	51
1761	559	162	4095	2334	.5699	143	.2558	1	4	62	17
3107	1485	309	5793	2686	.4636	200	.1346	1	12	25	3
1499	601	135	3536	2037	.5760	104	.1730	1	4	36	4
3447	1082	401	12172	8725	.7168	272	.2513	1	9	33	12
3050	973	328	6132	3082	.5026	236	.2425	9	9	67	11
6000	2523	834	11048	5048	.4569	341	.1351	1	5	5	32
958	468	129	2609	1651	.6328	78	.1666	1	2	33	31
1090	503	138	2741	1651	.6023	89	.1769	1	1	46	1
2172	871	272	4208	2036	.4838	145	.1664	1	3	14	10
3586	1624	470	8298	4712	.5678	325	.2001	54	54	154	33
313	100	56	909	596	.6556	52	.52	4	4	5	0
1300	712	194	2974	1674	.5628	129	.1811	1	18	10	5
1719	623	255	3504	1785	.5094	153	.2455	1	13	7	4
29	10	8	244	215	.8811	6	.6	1	1	0	1
14	6	8	232	218	.9396	4	.6666	1	1	0	0
261	108	24	639	378	.5915	17	.1574	1	1	2	3
659	302	71	1895	1236	.6522	25	.0827	1	1	13	2
307	137	88	873	566	.6483	226	.1897	1	2	1	1
114	57	24	1114	1000	.8976	9	.1578	3	7	1	0
450	236	110	1682	1232	.7324	31	.1313	21	27	25	2
133	67	29	449	316	.7037	16	.2388	1	1	0	1
287	133	96	1059	772	.7289	8	.0601	1	1	0	0
1202	732	403	2363	1161	.4913	88	.1202	2	2	2	0
266	142	44	836	570	.6818	23	.1619	3	3	0	0
225	139	7	730	505	.6917	11	.0791	10	13	0	3

139	63	37	746	607	.8136	15	.2380	1	2	0	0
360	279	1	1179	819	.6946	116	.0573	1	1	3	1
135	87	34	546	411	.7527	9	.1034	1	1	0	0
1764	679	305	4397	2633	.5988	169	.2488	20	23	47	9

Table 1. Results of Case Study Data Collection

Defect data from one system release representing a calendar interval of 6 months was collected for the 30 modules. The total number of defects captured for the 30 studied modules was 237. Table 1 depicts the results of the data collection. Each row of the table contains information for one of the studied modules. The first 11 columns correspond to the values of the candidate code-based metrics. The last column contains the total number of errors detected in the module.

In order to determine which of the predictors was most capable of identifying error prone modules, it was necessary to perform a data correlation. A number of possible correlation methods were analyzed to determine which would best be suited to the task. However, one overriding concern strongly directed the type of correlation method that had to be used. Without being able to make strong assumptions regarding the underlying relationship between the predictors and the error prone modules, it would not be possible to use any particular type of correlation (such as *linear* correlation). Therefore, it was decided that a **ranked** correlation would be the most useful, since it would not rely on any particular underlying relationship between the data points. In particular, in using ranked correlations, the actual data values themselves are not used. Instead, each data value is replaced with its ordinal rank, indicating its position relative to the other data values.

In particular, the *Spearman coefficient of rank correlation* was chosen [6]. This coefficient is calculated as:

$$R = 1 - \frac{6\Sigma D_i^2}{n(n^2-1)}$$

where n is the number of data pairs, and D_i is the signed difference between any two ranked data pairs. This formula results in a value between -1.0 and +1.0, where +1.0 indicates a perfect direct correlation, -1.0 indicates a perfect indirect correlation, and 0.0 indicates no correlation.

Choosing a degree of certainty of 99.9% for the data analysis of 30 data pairs, the value of the Spearman coefficient R would need to exceed +0.55 to show a direct correlation (one value increasing makes the other value increase), or would have to be less than -0.55 to show an indirect correlation (one value increasing makes the other value decrease).

Table 2 depicts the results of the Spearman R coefficients obtained by correlating the various predictor metrics against the number of errors in each module.

Predictor	Against Errors
NCSS	.66
SS	.77
CPLX	.67
LINES	.77
COMM	.77
COMMD	-.62
MCCB	.79

MCCBD	.14
ENTE	.16
ENTT	.50
PARM	.77

Table 2. Results of Case Study Data Analysis

It is interesting to note that all 3 of the volume measures performed quite well in their ability to predict the number of errors. Both **SS** and **LINES** scored 0.77, and **NCSS** scored 0.66. This is in direct agreement with popular belief that bigger modules will encounter more defects.

The 0.79 score for the McCabe measure was the highest correlation coefficient achieved in the case study, placing significant value in the cyclomatic complexity measure. Note, however, that the McCabe density measure (**MCCBD**) performed quite poorly (0.14). It seems that, in this case, raw complexity is significantly more important than the ratio of complexity to program size.

Another interesting note regarding the McCabe metric is its relationship to the volume measures. When the values obtained for **MCCB** were correlated against the values of **NCSS**, **SS**, and **LINES**, values of 0.97, 0.93, and 0.94 were obtained, respectively. This shows that complexity is almost perfectly related to program size (as previously reported by others [7]).

In studying the results achieved for "lines of comments" (**COMM**), an interesting anomaly occurred. Counterintuitively, the **COMM** measure scored a 0.77, indicating a **direct** relationship between the number of lines of comments and the defect count. That is, as the number of lines of comments increased, so did the defect count. This is counter to the normal belief that more comments will result in a more-maintainable module (less defects). To attempt to explain this anomaly, the values obtained for **COMM** were correlated against the values of **NCSS**, **SS**, and **LINES**. Correlation coefficients of 0.95, 0.91, and 0.99 were obtained, respectively. This shows that the number of lines of comments is almost perfectly related to program size - bigger programs have more comments. Since it was already shown that bigger programs contained more defects, the anomaly is explained.

Intuitively, the comment density measure (**COMMD**) received a correlation coefficient of -0.62, indicating an indirect relationship to maintainability. This relates well to the belief that having a higher percentage of comments in the code makes the code more maintainable.

The last indicator which performed well is the count of parameters measure (**PARM**) which received a score of 0.77. However, a correlation of the **PARM** measure to the 3 volume measures **NCSS**, **SS**, and **LINES** resulted in scores of 0.83, 0.75, and 0.84, indicating that the number of declared parameters is closely linked to the size of the module.

Conclusions and Future Research

Software maintenance has become the primary consumer of today's software development dollar, and maintenance costs continue to grow even larger. With so much money being spent on maintenance, the existence of a viable maintainability predictor would be highly valuable to companies interested in minimizing development costs. Prediction systems capable of highlighting unmaintainable code, or even poorly-implemented maintenance changes, would be invaluable.

Examining the results from this specific case study, it is interesting to note that the best predictive metric (and that which is easiest to collect) is simply "program size." In the environment of the case study, this would tend to indicate that programmers, who are interested in reducing

future corrective maintenance costs, should try to implement their modules to be as small as possible. Also, modules should probably be rewritten and decomposed into multiple, smaller modules when they begin to get too large. From a testing point of view, system testers should concentrate their "hot spot" testing on larger modules, since they are likely to contain more defects.

Although this case study was performed in one, particular development environment, the results achieved have shown that it is possible to predict maintainability. It is interesting to view the specific results and examine which metrics were best-capable of predicting maintainability, but the fact that one predictive metric functioned better than some other metric is not nearly as important as that something worked. Different predictors might function better in different development environments.

The bottom line is that development organizations should use some form of maintainability prediction system. Such a system should be viewed as another vital, indispensable tool in the programmer's tool set - one which offers significant potential for reducing the overall cost of software development.

This paper has described an approach for identifying error-prone modules in large evolutionary development environments. The approach consists of developing code-based, documentation-based and history-based measures. Our initial results show that simple and practical code-based measures may work quite well. Future research must be performed to assess the performance of the measures for other projects. Additional research must also be performed to assess the ability of the documentation-based and history-based measures to predict error-prone modules. Finally, the combination of all 3 sets of measures must be assessed to see if it is possible to increase the reliability of the prediction process. These investigations are currently underway and will be presented in future publications.

References

[1] McCabe, T. A complexity measure. *IEEE Transactions on Software Engineering* SE-2, 12 (December 1976), 308-320.

[2] Halstead, M. *Elements of software science.* Elsevier North-Holland, New York, 1977.

[3] Arthur, L. *Programmer productivity: Myths, methods, and murphy's law.* John Wiley and Sons, New York, New York, 1983.

[4] Frost, D. Software maintenance and modifiability. In *Proceedings of the Phoenix Conference on Computers and Communications 1985* (Scottsdale, Arizona, March 20-22, 1985), 489-494.

[5] Liu, C. A look at software maintenance. *Datamation.* 22, 11 (November, 1976), 51-55.

[6] Gibbons, J. *Nonparametric methods for qualitative analysis.* Holt, Rinehart, and Winston, Tuscaloosa, Alabama, 1976.

[7] Henry, S., Kafura, D., and Harris, K. On the relationships among three software metrics. In *1981 ACM Workshop / Symposium on Measurements and Evaluation of Software Quality* (College Park, Maryland, March 25-27, 1981), 81-88.

A Systematic Approach to Regression Testing

Abstract

With increased software complexity and size, more resources are being allocated for testing during the development and maintenance phases. Typically, modifications of the program and requirements are followed by testing and debugging activities, which are a major contribution to the high cost of evolving software systems. In order to reduce this cost, functional and structural changes must be tested in an efficient manner. Regression testing describes the selective testing that is carried out to ensure that no adverse side effects have been introduced and that the original requirements are still met. This paper describes a systematic approach for reducing the cost and time involved in regression testing through the use of linear programming techniques. They can be applied to both programs and requirements to minimise the number of tests for rerun after modifications have been made and can provide the means for rapidly assessing the impact of program alterations.

Biographical Sketch

J.Hartmann :

He is a PhD candidate in the Centre for Software Maintenance at the University of Durham. His research interests include issues relating to software testing and maintenance. Hartmann received a BS in Computing/Electronics and an MS in Microelectronics from Durham University. He is a member of ACM and IEEE.

D.J. Robson :

He is a lecturer in Computer Science at the University of Durham. His research interests include software maintenance and software testing. Robson received a BS and MS in Pure Mathematics from Sheffield University and an MS and PhD in Computer Science from the University of Hull. He is a member of the British Computer Society, IEEE Computer Society, and ACM.

Authors' Mailing Address

Mr Jean Hartmann,
Dept. of Computer Science,
Science Laboratories,
University of Durham,
Durham DH1 3LE,
United Kingdom.

Tel : +44-91-374-3658
Fax: +44-91-374-3741
E-mail: J.Z.W.Hartmann@uk.ac.durham

1. Introduction

Evolving software systems undergo numerous modifications during their development and maintenance. Modifications are made for a number of reasons. They include enhancements to improve performance and portability, corrections to errors found during previous test sessions, and the development of new system features.

Past surveys have indicated that approximately 50% of the programming effort is spent in the maintenance phase of the software lifecycle [24]. As larger and more complex software systems are developed, the probability of introducing side effects during maintenance increases. A study by Collofello and Buck [7] found that as many as 72% of the revealed defects were related to side effects introduced into the unchanged portions of a system.

Regression testing is the selective testing that ensures that no adverse side effects have been introduced into the system and which verifies that the system still meets its requirements [1]. Current regression testing strategies suggest the rerunning of a) a set of 'confidence tests', which exercise the main system functionalities, b) a set of test cases which invoke the modified modules, c) a set of test cases chosen by experienced staff, and d) the entire test suite.

In general, regression testing is performed numerous times throughout the product's lifecycle, and the effort and cost spent on it can exceed initial development testing. Thus, it is important to develop an efficient test selection technique for regression testing. It must focus on the affected parts of the requirements or program, selecting only the relevant set of test cases from the existing test suite. Most organisations have automated their regression testing procedure by means of interactive capture-replay tools [23, 18]. Although these tools can provide a significant saving in testing effort, they do not yet provide efficient mechanisms for automatically determining the subset of test cases that need to be rerun after maintenance.

This paper outlines a prototype software environment to evaluate a test selection strategy based on linear programming. It allows the impact of modifications to be assessed in terms of the minimum number of tests that need to be rerun. Furthermore, the prototype can predict test duration and cost required in revalidating the system, if the user can quantify the time and cost it takes to create and execute individual tests. The remainder of the paper is organised as follows: Section 2 gives a brief overview of the current state of the art; Section 3 outlines the implementation of the prototype; Section 4 explains the test selection procedure by way of an example; Sections 5 and 6 summarise some relevant background information on program analysis and linear programming; with concluding remarks and future work presented in the last section.

2. Test Selection Strategies

2.1 Background

Current test selection strategies determine a subset of tests by first analysing a program's control and data flow. A directed graph called the *control flow graph* is used to represent the program under consideration. Each *node* in the graph represents a *basic block* or *segment*, defined as a sequence of statements where, if the first statement is executed, all subsequent statements are also executed. Each *edge* in the graph represents the transfer of control between nodes. Thus, a *path* refers to a finite sequence of nodes connected by edges, whose first node is an entry node and whose last node is an exit node. A *simple path* is a path which traverses a program loop in only a few iterations. For data flow, three basic actions are associated with program variables, namely *def*, when there is variable assignment, *ref*, when a variable value is used, and *undef*, where the variable is unavailable. Based on these actions, Rapps and Weyuker [34] describe a family of test coverage criteria.

2.2 Ostrand and Weyuker

Ostrand and Weyuker [31] present their technique as an extension to the ASSET data flow testing tool [9]. ASSET requires users to specify a coverage criterion for the program under consideration, tests are then executed with the tool indicating the percentage coverage that is achieved. Essentially, data flow testing is based on *definition-use associations*, (D-U), in which a program path traversed by a test case exercises a variable definition and its subsequent use without an intervening redefinition. D-U pairs are defined by a triple, (x,y,z), where x represents the variable, and y and z are the nodes where the variable is defined and used, respectively. For regression testing purposes, the authors propose that after maintenance only the newly-created and definition-use pairs based on modified variables need to be rerun. Harrold [12] discusses a similar approach based on incremental data flow analysis. She extends the control flow graph to support testing history information. For each definition in a node of the graph, the nodes and edges that use the definition are attached. In addition, a list containing the nodes where each variable is defined, and a list of all definition-use pairs that are used to meet the coverage criterion, are maintained. Interprocedural data flow testing is also described, where definition-use pairs across subroutine calls are analysed. Another approach to test selection by incremental analysis is presented by Taha *et al.* [36]. They also propose ways in which to use the data flow relations to generate new test cases and outline a strategy for fault localisation and repair.

2.3 Leung and White

Leung and White [21] describe a test selection method in which a bit vector is associated with each node in the control graph. If a test case *i* traversed a node, then the *ith* bit of the node's bit vector is set to one. When a node is modified, deleted, or split, the corresponding bit vector determines the test cases that need to be rerun. The authors also relate test selection to test suite maintenance, which is concerned with developing new test cases as well as partitioning existing tests into three categories, namely *reusable* tests that exercise unmodified portions of the code, *retestable* tests covering those parts of the program that have been modified and *obsolete* tests which constitute tests which are no longer required and can thus be deleted from the suite [22].

2.4 Fischer

Fischer *et al.* [8] describe a test selection methodology which uses linear optimisation to determine a minimum set of test cases to rerun. The technique is based on a directed graph representation of the program. *Connectivity*, the direct transfer of control between nodes, and *reachability*, the indirect transfer of control, are calculated. Furthermore, variable definitions and uses are computed for each node in the graph. A set of simple paths is mapped against nodes to represent the test coverage. After modifications, the reachability and affected variable references for each modified node are determined. A *zero-one integer programming* model is formulated and by solving it a set of tests is selected. This set represents the minimum number of test paths that need to be executed to ensure that all nodes, which directly and indirectly interact with the modified nodes, are revalidated.

2.5 Yau and Kishimoto

In the test selection strategy proposed by Yau and Kishimito [41] a test suite is developed from the program specification and code. Initially, input partitions are derived from a cause-effect graph that represents the specification, and the *reaching set* for each node in the program graph is calculated. A reaching set is the set of all possible paths that start at the entry node and end at the particular node. Test cases are symbolically executed and allocated to the different input partitions. To satisfy the test coverage criterion, at least one test case must cover each input partition. Program alterations result in the technique deriving input

partitions for the modified program, and the existing test suite is examined to verify that the testing criterion is still satisfied. If not, new test cases need to be generated by the user. Test cases from the existing suite are selected based on whether they follow paths in the reaching set of the modified nodes.

3. *RETEST* - A Test Selection Prototype

3.1 Introduction

A prototype for the evaluation of a test selection strategy based on linear programming is currently being implemented to investigate its effectiveness. The main objective is to assess the impact of modifications on the number of test cases that need to be rerun, where the test coverage is based on control or data flow criteria. Users must select a coverage criterion and can input cost and time estimates for creating and executing individual test cases. Once the prototype has completed its analysis of the program under consideration, the user needs to generate a set of tests to satisfy the chosen coverage criterion. To simplify the implementation and to develop a prototype in a short time period, the incremental analysis and updating mechanisms are not included. This means that users need to specify the type of changes made during maintenance and the prototype will then select a minimal set of tests to ensure that the affected program parts are exercised.

3.2 An Overview

A block diagram of the prototype known as the **RE**gression **TE**sting **S**upport **T**ools (RETEST) is shown in Figure 1. The annotator instruments syntactically correct C programs. Before parsing any source code, it invokes the C preprocessor to expand any macros or header files found in the code. If during parsing, any syntax errors are encountered, the tool will highlight these errors and advise users to correct them before resubmitting the code for analysis.

Both the control and data flow analysers provide an inter- and intraprocedural analysis, where the control flow within each C function is abstracted into an *intra*procedural flow graph and the call-graph is generated to represent the *inter*procedural flow. Thus, segment, branch, or decision-to-decision (DD-path) information is prepared, depending on users requirements. Alternatively, the definition-use associations are generated when the user specifies a data flow criteria. A simple database has been integrated into the environment to store the resulting control and data flow information.

As the prototype is based on a path selection strategy, all test requirements are mapped against a set of simple paths resulting in a *test coverage frequency matrix*. Initially, the path generation tool produces a set of expressions consisting of *elementary paths* and *simple loop patterns* [39]. These are expanded to form a set of unique paths in which the number of loop iterations is, by default, set to one, but which can be relaxed to two or more, if necessary.

Once a test coverage frequency matrix has been formed, users must execute test data in order to complete the given matrix. It is recommended that tests should be executed in an ordered fashion. This entails executing functional, or black-box, tests first and then subsequently supplementing the test suite with structural, or white-box, test cases. The analyser will display the coverage that is achieved, and any paths and associated measures that are still uncovered.

Due to the lack of incremental analysis tools, RETEST requires users to identify those areas of code that are *directly* affected during maintenance, by specifying the C functions and segment numbers that have altered. The integer programming model consists of two parts,

where the first part represents the type of reduction that is to be achieved, for example, choosing the smallest number of tests that need to be rerun, selecting tests to run in the shortest time period, or for the lowest cost. The other is derived from the test coverage frequency matrix and a coverage requirement array. Further details on integer programming and its terminology are given in Section 6. Once completely specified, the model is solved and the resulting test cases are displayed.

The current test selection prototype is UNIX™ based and has been developed in C. The Unix development tools, Lex and Yacc, were used in generating the parsers. At present, the different tools are invoked using a command-line interface, but a graphical user interface based on X Windows may be added later. By designing the prototype in a modular fashion, individual components can be extended and used as separate testing and maintenance tools such as coverage analysers, cross-referencers, and ripple-effect analysers. As part of the prototype, it has also been possible to integrate a McCabe's complexity metric analyser and comments extractor [17].

™ UNIX is a trademark of AT&T.

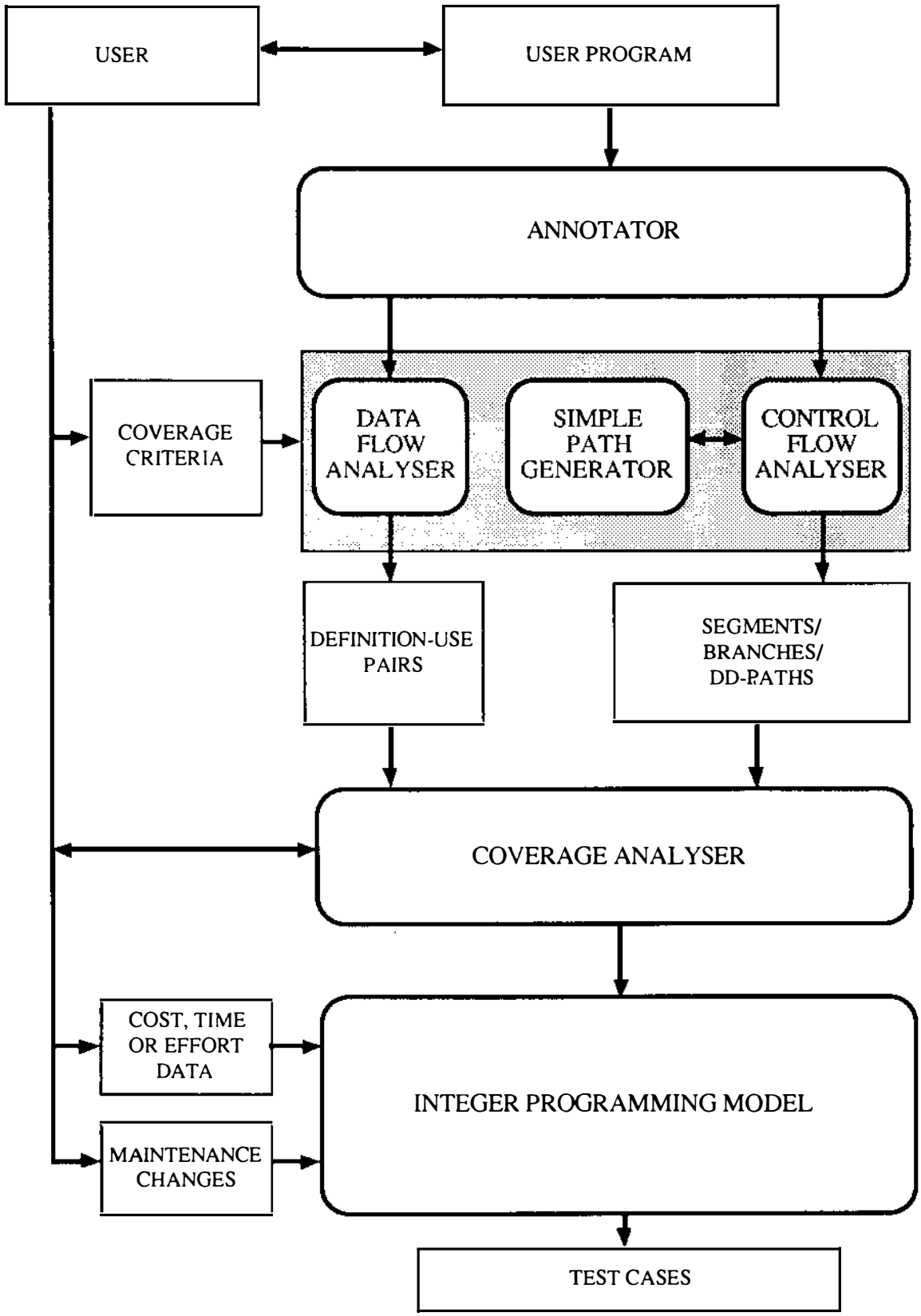


Figure 1 : A Block Diagram of the Prototype Implementation

4. Application of Test Selection

In the past, several authors have applied linear programming techniques to structural [33, 25] and functional [19] testing. Their concepts are very similar to the ones expressed in this paper, which applies them to regression testing [15]. Although the following discussion demonstrates test selection at the unit level, the strategy can be applied equally well to regression testing at the integration and system level[19]. For integration testing, the same test selection procedure is followed and the interprocedural flow analysis enables control and data flow test criteria to be applied across function boundaries [13].

Test coverage tools such as TCAT[28] and ASSET[9] are used in validating software based on control or data flow criteria, respectively. Our test selection strategy can be applied to test suites that have been established using either of these criteria. More specifically, the example given in Section 4.2, illustrates test selection based on node or statement testing, and the *all-uses* criterion[34].

The test selection technique described here is particularly useful when certain conditions hold:

- If the program requirements remain unchanged, all of the existing test cases are still valid after the modifications are made. This is known as *corrective regression testing* [22].

- The program being maintained has been validated so that the test coverage frequency matrix contains sufficient test paths to exercise, at least once, each test requirement.

4.1 Types of Changes

At the program level, modifications are made for a number of reasons. They include code enhancements to improve performance and portability, corrections to errors found during previous test sessions, and the development of new system features. However, in terms of the flow graph, they can typically be divided into two categories. They consist of:

- Structural changes, which represent modifications to the flow graph by the addition or deletion of a number of nodes and edges. As a result of changing the overall program structure, the tests selected for rerun will probably exercise both unmodified and new code. This will require users to generate additional test cases to satisfy the coverage criteria, based on the prototype providing an indication of any uncovered paths.

- Non-structural changes, which pertain to alterations that are made to variables within the nodes. Thus, the syntactic structure of the modified program flowgraph is identical to that of the original, while the computations within some nodes are different. In this case, program changes may cause the selected tests to traverse different paths, resulting in new tests being derived to exercise some of the existing paths, and requiring others to be deleted from the test suite, as they are now structurally redundant. Once again, the prototype will indicate which paths have been left uncovered and which paths are traversed by several test cases.

4.2 Test Selection Procedure

To illustrate the steps involved in the test selection procedure, a piece of C code is to be maintained, which has been previously tested using two different test requirements, namely a control and a data flow criterion. The sample code and its corresponding control flow graph are shown in Figure 2a-b. The purpose of the C function is to calculate x to the power y, where both x and y are integers.

Table 1 shows the test coverage frequency matrix after applying the node or statement testing strategy. In it, each path represents a test case which exercises a set of nodes, zero, once, or several times. For example, test path **P1** traverses nodes 1,2,3,5,6,8,9,10. Similarly, Table 2 illustrates the test coverage frequency matrix after the all-uses criterion has been satisfied. Here, the test paths exercise the different definition-use pairs that have been identified during an earlier program analysis. For example, test path **P3** traverses definition-use pairs (y,1,2), (y,1,4), (y,1,8), (p,4,6), (z,5,10), (z,5,9), and (z,9,10).

In our test selection strategy, program modifications are defined as a series of elementary changes to nodes, where a modification to node 7, as shown in Figure 2c, results in a deletion of the old code, followed by an insertion of new code. When considering a series of changes to different nodes in the flowgraph, each modification has to be separately analysed and retested. With RETEST, the user needs to also specify the node number(s) corresponding to the modification(s).

In the selection of tests based on node coverage, the coverage requirement array, shown in Table 1 as the last column, indicates those nodes that are directly or indirectly affected by the modified node. This information is extracted from the reachability matrix (not shown here), which is generated during the control flow analysis. By logically ORing the rows and columns corresponding to the modified node, the reachability for individual nodes is calculated. For the data flow criterion, the coverage requirement array consists are values which correspond to those definition-use associations in which either the definition node, or the use node has been modified. The resulting values are given in the last column of Table 2.

Based on the information provided in Tables 1 and 2, two integer programming models can be formulated, one for each test requirement. To simplify the models, a set of reduction rules can be applied in each instance.

- Remove all test cases that do not traverse the modified node, or affected definition-use pairs.
- Eliminate rows, which are duplicates, contain all test cases represented in the test suite, and have zeroes for the corresponding coverage requirement.

The final, reduced models and their solutions are illustrated in Figure 3.

Z	=	x_1	$+x_2$	$+x_3$	$+x_4$	$+x_5$	$+x_6$	$+x_7$	$+x_8$		
3:						x_5	$+x_6$			$\geq$	1
4:								x_7	$+x_8$	$\geq$	1
7:						x_5	$+x_6$	$+x_7$	$+x_8$	$\geq$	1
9:						x_5		$+x_7$		$\geq$	1

Solution : $Z = 2$, where Test Cases (x_5,x_7) have been selected to be rerun
Figure 3a : Integer Programming Model for Node Test Selection

$Z =$	x_1	$+x_2$	$+x_3$	$+x_4$	$+x_5$	$+x_6$	$+x_7$	$+x_8$	$+x_9$	$+x_{10}$	$+x_{11}$	$+x_{12}$	
(P,3,7):					x_5	$+x_6$			$+x_9$	$+x_{10}$			$\geq$ 1
(P,4,7):							x_7	$+x_8$			$+x_{11}$	$+x_{12}$	$\geq$ 1
(P,7,7):									x_9	$+x_{10}$	$+x_{11}$	$+x_{12}$	$\geq$ 1
(Z,5,7):					x_5	$+x_6$	$+x_7$	$+x_8$	$+x_9$	$+x_{10}$	$+x_{11}$	$+x_{12}$	$\geq$ 1
(Z,7,7):					x_5		$+x_7$		$+x_9$		$+x_{11}$		$\geq$ 1

Solution : $Z = 2$, where Test Cases (x_7,x_9) have been selected to be rerun
Figure 3b : Integer Programming Model for All-Uses Test Selection

```
void example()
{
 int x,y,p;
 float z;

1  scanf("%d%d",&x,&y);

2  if(y < 0)
   {
3    p = -y;
   }
  else
   {
4    p = y;
   }

5  z = 1.0;

6  while(p != 0)
   {
7    z *= (float) x;
     p--;
   }

8  if(y < 0)
   {
9    z = (1/z);
   }

10 printf("%f\n",z);
}
```

Figure 2a : Sample C Code

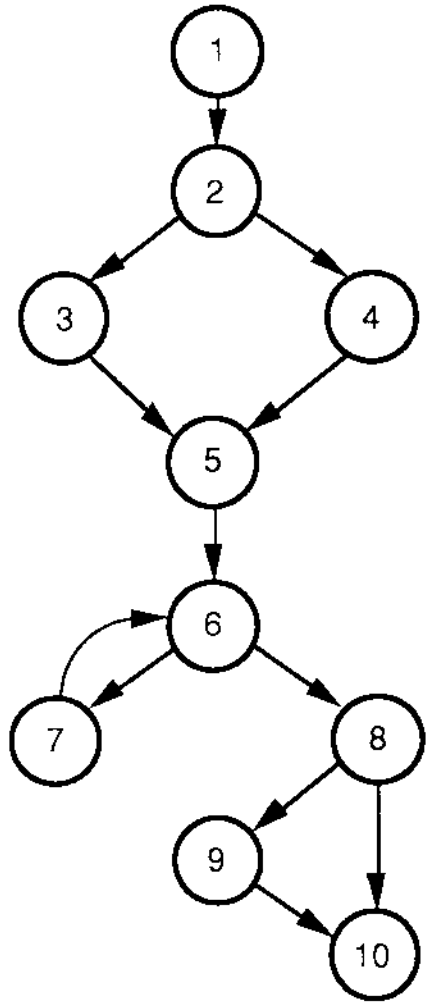


Figure 2b : The Control Flow Graph

```
6  while(p != 0)
   {
7.1  if(p % 2)
     {
7.2    z *= (float) x;
     }

7.3  p -= 2;
     x *= 2;
   }
```

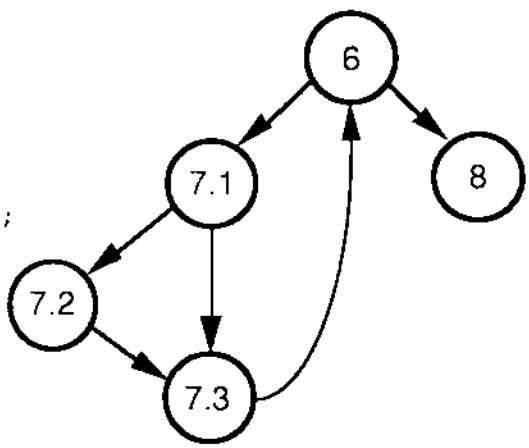


Figure 2c : The Modifications

Path / Node	P1	P2	P3	P4	P5	P6	P7	P8	Reach-ability
1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1
3	1	1	0	0	1	1	0	0	1
4	0	0	1	1	0	0	1	1	1
5	1	1	1	1	1	1	1	1	1
6	1	1	1	1	2	2	2	2	1
7	0	0	0	0	1	1	1	1	1
8	1	1	1	1	1	1	1	1	1
9	1	0	1	0	1	0	1	0	1
10	1	1	1	1	1	1	1	1	1

Table 1 : Test Coverage Frequency Matrix and Coverage Requirements for Nodes

Path / D-Us	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	Affected D-Us
(X,1,7)	0	0	0	0	1	1	1	1	1	1	1	1	1
(Y,1,2)	1	1	1	1	1	1	1	1	1	1	1	1	0
(Y,1,3)	1	1	0	0	1	1	0	0	1	1	0	0	0
(Y,1,4)	0	0	1	1	0	0	1	1	0	0	1	1	0
(Y,1,8)	1	1	1	1	1	1	1	1	1	1	1	1	0
(P,3,6)	1	1	0	0	1	1	0	0	1	1	0	0	0
(P,4,6)	0	0	1	1	0	0	1	1	0	0	1	1	0
(P,3,7)	0	0	0	0	1	1	0	0	1	1	0	0	1
(P,4,7)	0	0	0	0	0	0	1	1	0	0	1	1	1
(P,7,7)	0	0	0	0	0	0	0	0	1	1	1	1	1
(Z,5,7)	0	0	0	0	1	1	1	1	1	1	1	1	1
(Z,5,10)	1	1	1	1	1	1	1	1	1	1	1	1	0
(Z,5,9)	1	0	1	0	1	0	1	0	1	0	1	0	0
(Z,7,7)	0	0	0	0	0	0	0	0	1	1	1	1	1
(Z,7,9)	0	0	0	0	1	0	1	0	1	0	1	0	1
(Z,7,10)	0	0	0	0	1	1	1	1	1	1	1	1	1
(Z,9,10)	1	0	1	0	1	0	1	0	1	0	1	0	0

Table 2 : Test Coverage Frequency Matrix and Coverage Requirements for All-Uses

5. Inter- and Intraprocedural Analysis

This section summarises some background information on control and data flow analysis. Both theoretical and practical issues are discussed.

5.1 Control Flow

5.1.1 Connectivity and Reachability

When considering graphs, the direct and indirect flow between nodes needs to be calculated. The former is referred to as the *connectivity* and the latter as the *reachability*. For example, if node **a** was directly connected to **b**, and node **b** was connected to node **c**, then node **a** is indirectly connected to **c**. To compute the reachability of a specific node using its connectivity requires a transitive closure algorithm. Many examples of such algorithms exist, including the Warshall algorithm[37].

5.1.2 Path Generation

As mentioned earlier, the prototype is based on a selection strategy in which test requirements are mapped against a set of simple paths. The resulting test coverage frequency matrix includes elementary paths and paths in which loops are visited at least once[30]. Although structural testing using a graphical path has its weaknesses and may contain numerous infeasible paths[16]. Lin and Chung [25] propose an iterative solution to this problem. They suggest solving the integer programming model with the user identifying any infeasible paths in the solution. These paths can then be removed from the existing test suite and the model resolved. Subsequently, the above steps are repeated until all paths in the solution are known to be feasible. In our test selection strategy, we follow a similar approach.

5.2 Data Flow

In the past, many publications have addressed the subject of data flow and its problems. As programming languages became more sophisticated, concepts such as information hiding, data abstraction, and dynamic data structures were introduced. Subsequently, authors adapted their methods to address problems with pointer aliasing, function call side-effects, and reference parameters [4,29,2,38]. However, most methods provide only approximate algorithms for resolving these problems. Currently, the inter- and intraprocedural data flow analysis of C software is under investigation [32].

Data flow testing criteria are based on the analysis of variable references in a program. These references are identified by the variable's name and definition locations. When a variable name denotes multiple references, however, they can be difficult to track. This problem appears, in particular, when arrays and pointers are used. Ideally, each array element should be treated as a separate variable. However, it is usually not possible to statically determine the specific element to which an array occurrence refers. The simplest solution is to treat each definition or use of an array element as an access of the whole array. Some research has been conducted to investigate these data flow problems[6]. With pointer variables, a new dummy variable is created for every level of indirection implied in their declaration. During parsing, all references to the original pointer variable are then analysed and allocated to one of the associated variables, if necessary. The RETEST prototype is based on work conducted by Xue[40].

5.3 Incremental Analysis and Updating

A significant task, that has not been addressed or implemented, in the RETEST prototype, is the construction of an underlying, preferably incremental mechanism, for compiling and updating the differences in testing history, and control and data flow between test sessions. Moreover, such an analysis needs to be performed at both the inter- and intraprocedural level. Most advances in this area can be attributed to Ryder [27, 35], and accurately recalculating data flow dependencies at the interprocedural level has not yet been achieved [5]. Therefore, the prototype will initially be enhanced so that, between test sessions, any program differences are identified via an exhaustive control and data flow analysis rather than an incremental analysis.

6. Integer Programming

Every integer programming model consists of two parts.

The first part represents the type of reduction that is to be achieved, namely the minimum number of tests, or the minimum effort, or cost required to rerun the tests. This is known as the *objective function*, **Z**. An objective function can either be minimised, or maximised. For test selection, the objective function is to be minimised. Each algebraic term expressed in **Z** relates to a test case from the existing test suite and a coefficient that represents a weighted cost or time factor.

The second part is derived from the test coverage frequency matrix, and a set of integer values, known as the *coverage requirement array*. Together they represent a series of *constraints* that are placed upon the objective function. When considering control flow criteria, the coverage requirement array reflects the reachability of the modified segment or branch. In terms of data flow, the array represents the definition-use associations that have been affected by the modifications.

Mathematically, the class of problem that is being defined by the test selection strategy is known as *set covering* [3]. Theoretically, it is considered to be NP-complete[10] which indicates that there is no definitive algorithm to solve such problems in a reasonable amount of time. However, in practice, implicit enumeration techniques and heuristics can be applied[26]. Two important goals for optimisation are to reduce the overall size of a given model and improve the efficiency of the solving algorithms. For the former, a set of reduction rules can be used[11], while the latter considers implicit enumeration algorithms such as those discussed by Lemke *et al.* [20].

An alternative use of integer programming is in the selection of a representative set of test cases to exercise system requirements[19]. This is achieved by identifying redundant test cases and therefore reducing the overall size of test suites. Preliminary research on test suite reductions, based on heuristics, has been conducted by Harrold *et al.*[14].

7. Conclusion and Future Research

At present, the RETEST prototype is being evaluated using a suite of C programs, which vary in size and program complexity. With the prototype, the impact of modifications is being assessed in terms of the minimum number of tests that need to be rerun. By including the time, cost and effort spent executing individual tests, regression test duration, effort, and cost can be predicted. To illustrate the effectiveness of different test coverage criteria for regression testing, we can select different criteria, define a set of maintenance changes, and use the prototype to choose those test cases that need to be rerun. We regard our prototype more as a support environment, which would be used to complement any regression testing tools.

Future research will be directed towards refining and extending the prototype environment. In particular, the problems associated with the inter- and intraprocedural data flow analysis of C need to be investigated in more detail. Moreover, an incremental analysis tool needs to be defined and implemented.

8. Acknowledgements

Jean Hartmann is sponsored by British Telecom Research Centre, Martlesham Heath, Ipswich, IP5 7RE, U.K. and would like to thank them for their support. He would also like to acknowledge the referees for providing some useful comments on the paper.

9. References

[1] *"IEEE Standard Glossary of Software Engineering Terminology"*. IEEE Computer Society Press, Los Alamitos, 1983.

[2] Allen, F.E. and Cocke, J. "A Program Data Flow Analysis Procedure", *Communications of the ACM*, vol. 19, no. 3, pp. 137-47, March, 1976.

[3] Balas, E. and Padberg, M.W. "Set Partitioning". In *Combinatorial Programming: Methods and Applications*, Reidel Publishing, ed. Roy, B., pp. 205-58, 1974.

[4] Barth, J.M. "A Practical Interprocedural Data Flow Analysis", *Communications of the ACM*, vol. 21, no. 9, pp. 724-36, September, 1978.

[5] Burke, M.G. and Ryder, B.G. "A Critical Analysis of Incremental Iterative Data Flow Analysis Algorithms", *IEEE Transactions on Software Engineering*, vol. SE-16, no. 7, pp. 723-8, July, 1990.

[6] Calliss, F.W. and Cornelius, B.J. "Dynamic Data Flow Analysis of C Programs". In *Proceedings of 21st Annual Hawaii International Conference on System Sciences (HICSS)*, IEEE Computer Society Press, Los Alamitos, pp. 518-23, January, 1988.

[7] Collofello, J.S. and Buck, J.J. "Software Quality Assurance for Maintenance", *IEEE Software*, vol. 4, no. 5, pp. 46-51, September, 1987.

[8] Fischer, K.F., Raji, F., and Chruscicki, A. "A Methodology for Re-Testing Modified Software". In *Proceedings of National Telecommunications Conference*, IEEE Computer Society Press, Los Alamitos, pp. B6.3.1-6, November, 1981.

[9] Frankl, P.G., Weiss, S.N., and Weyuker, E.J. "ASSET: A System to Select and Evaluate Tests". In *Proceedings of International Conference on Software Tools*, IEEE Computer Society Press, Los Alamitos, pp. 72-9, April, 1985.

[10] Garey, M.R. and Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Company, 1979.

[11] Garfinkel, R.S. and Nemhauser, G.L. *Integer Programming*. John Wiley, Interscience, 1972.

[12] Harrold, M.J. "An Approach to Incremental Testing", Technical Report 89-1, University of Pittsburgh, 1988.

[13] Harrold, M.J. and Soffa, M.L. "Efficient Computation of Interprocedural Data Dependencies". In *Proceedings of International Conference on Computer Languages*, IEEE Computer Society Press, Los Alamitos, pp. 297-306, March, 1990.

[14] Harrold, M.J., Gupta, R., and Soffa, M.L. "A Methodology for Controlling the Size of a Test Suite". In *Proceedings of Conference on Software Maintenance (CSM-90)*, IEEE Computer Society Press, Los Alamitos, pp. 302-10, November, 1990.

[15] Hartmann, J. and Robson, D.J. "Techniques for Selective Revalidation", *IEEE Software*, vol. 7, no. 1, pp. 31-6, January, 1990.

[16] Hedley, D. and Hennell, M.A. "The Causes and Effects of Infeasible Paths in Computer Programs". In *Proceedings of International Conference on Software Engineering (ICSE)*, IEEE Computer Society Press, Los Alamitos, pp. 259-66, August, 1985.

[17] Kuhn, D.R. "Static Analysis Tools for Software Security Certification". In *Proceedings of 11th National Computer Security Conference*, pp. 290-8, 1988.

[18] Leach, D.M., Paige, M.R., and Satko, J.E. "AUTOTESTER: A Testing Methodology for Interactive User Environments". In *Proceedings of 2nd Annual Phoenix Conference on Computers and Communications*, IEEE Computer Society Press, Los Alamitos, pp. 143-7, March, 1983.

[19] Lee, J.A.N. and He, X. "A Methodology for Test Selection", *Journal of Systems and Software*, vol. 13, no. 3, pp. 177-85, November, 1990.

[20] Lemke, C.E., Salkin, H.M., and Spielberg, K. "Set Covering by Single-Branch Enumeration with Linear-Programming Subproblems", *Operations Research*, vol. 19, no. 4, pp. 998-1022, July/August, 1971.

[21] Leung, H.K.N. and White, L. "A Study of Regression Testing", Technical Report TR 88-15, University of Alberta, September, 1988.

[22] Leung, H.K.N. and White, L.J. "Insights into Regression Testing". In *Proceedings of Conference on Software Maintenance(CSM-89)*, IEEE Computer Society Press, Los Alamitos, pp. 60-9, October, 1989.

[23] Lewis, R., Beck, D.W., and Hartmann, J. "Assay - A Tool to Support Regression Testing". In *Proceedings of 2nd European Software Engineering Conference (ESEC'89)*, pp. 487-96, September, 1989.

[24] Lientz, B.P. and Swanson, E.B. *Software Maintenance Management*. Addison-Wesley, 1980.

[25] Lin, J.C. and Chung, C.G. "Zero-One Integer Programming Model in Path Selection Problem of Structural Testing". In *Proceedings of COMPSAC'89*, IEEE Computer Society, Los Alamitos, pp. 618-27, September, 1989.

[26] Müller-Merbach, H. "Modelling Techniques and Heuristics for Combinatorial Problems". In *Combinatorial Programming: Methods and Applications*, Reidel Publishing, ed. Roy, B., pp. 3-27, 1974.

[27] Marlowe, T.J. and Ryder, B.G. "Hybrid Incremental Alias Algorithms". In *Proceedings of 24th Annual Hawaii Conference on System Sciences(HICSS)*, IEEE Computer Society Press, Los Alamitos, pp. 428-37, January, 1991.

[28] Miller, E. "Advanced Methods in Automated Software Test". In *Proceedings of Conference on Software Maintenance(CSM-90)*, IEEE Computer Society Press, Los Alamitos, pp. 111, November, 1990.

[29] Myers, E.W. "A Precise Inter-Procedural Data Flow Algorithm". In *Proceedings of 8th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pp. 219-30, 1981.

[30] Ntafos, S.C. and Hakimi, S.L. "On Path Cover Problems in Digraphs and Applications to Program Testing", *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 520-9, September, 1979.

[31] Ostrand, T.J. and Weyuker, E.J. "Using Data Flow Analysis for Regression Testing". In *Proceedings of 6th Annual Pacific NorthWest Quality Assurance Conference*, pp. 233-48, September, 1988.

[32] Ostrand, T.J. "Data Flow Based Testing Techniques". In *Proceedings of Eight Pacific Northwest Software Quality Conference*, pp. 218-27, October, 1990.

[33] Popkin, G.S. "*A Binary Programming Solution to a Problem in Computer Program Testing*", PhD. thesis, Polytechnic Institute of New York, Brooklyn, NY, January, 1987.

[34] Rapps, S. and Weyuker, E.J. "Selecting Software Test Data Using Data Flow Information", *IEEE Transactions on Software Engineering*, vol. SE-11, no. 4, pp. 367-75, April, 1985.

[35] Ryder, B.G. and Paull, M.C. "Incremental Data-Flow Analysis Algorithms", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 10, no. 1, pp. 1-50, January, 1988.

[36] Taha, A.B., Thebaut, S.M., and Liu, S.S. "An Approach to Software Fault Localization and Revalidation Based on Incremental Data Flow Analysis". In *Proceedings of COMPSAC'89*, IEEE Computer Society, Los Alamitos, pp. 527-34, September, 1989.

[37] Warshall, S. "A Theorem on Boolean Matrices", *Journal of the ACM*, vol. 9, no. 1, pp. 11-2, 1962.

[38] Weihl, W.E. "Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables". In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pp. 83-92, 1980.

[39] White, L.J. and Wiszniewski, B. "SILOP: A Tool for Path Testing of Computer Programs with Loops", Technical Report TR-CES-90-06, University of Alberta, April, 1990.

[40] Xue, B. "CPAT - A C Program Analysis Tool", Beijing Information Technology Institute, 1989.

[41] Yau, S.S. and Kishimoto, Z. "A Method for Revalidating Modified Programs in the Maintenance Phase". In *Proceedings of COMPSAC'87*, IEEE Computer Society Press, Los Alamitos, pp. 272-7, October, 1987.

On Testing Expert Systems Software

Guillermo A. Francia III and Andrew H. Sung
Department of Computer Science
New Mexico Tech
Socorro, NM 87801, U.S.A.
Internet: francia@nmt.edu, sung@nmt.edu
Tel: (505) 835-5209, Fax: (505) 835-6329

Abstract

Expert systems are becoming more complicated and finding more applications in various areas. Since failure of expert systems may result in costly losses, establishing quality assurance procedures to ensure the reliability of such systems is of great importance to their development.

Software quality assurance relies on testing and validation. Many of the traditional software testing techniques and tools, however, are not directly applicable to expert systems software, particularly those implemented in logic programming languages. To establish objective quality assurance procedures, formal criteria need to be defined for the testing of expert systems so that testing efforts may be measured and directed quantitatively. In this paper, a collection of testing criteria for expert systems is defined and analyzed. These criteria are designed specifically for testing the knowledge bases (KBs) of expert systems that are implemented in logic programming languages.

Biographical Sketch of Authors

Guillermo A. Francia III received his B.S. degree in Mechanical Engineering from Mapua Institute of Technology, Manila, Philippines, and his M.S. degree in Computer Science from New Mexico Tech, Socorro, New Mexico. Francia is currently working on his Ph.D. degree in Computer Science at New Mexico Tech. His research interests include expert systems, software testing and validation, numerical computation, and parallel processing.

Andrew H. Sung is associate professor and chairman of the Computer Science Department at New Mexico Tech. His research interests include parallel processing, software engineering, expert systems, and computational complexity. Sung received his Ph.D. from State University of New York at Stony Brook.

On Testing Expert Systems Software

Guillermo A. Francia III and Andrew H. Sung
Department of Computer Science
New Mexico Tech
Socorro, NM 87801, U.S.A.
Internet: sung@nmt.edu, francia@nmt.edu
Tel: (505) 835-5126, Fax: (505) 835-6329

Abstract

Expert Systems (ESs) are becoming more complicated and finding more applications in various areas. Since failure of expert systems may result in costly losses, establishing quality assurance procedures to ensure the reliability of such systems is of great importance to their development.

Software quality assurance relies on testing and validation. Most of the traditional software testing techniques and tools, however, are not directly applicable to ESs software, particularly those implemented in logic programming languages. For example, due to the lack of control–flow and data–flow concepts in Prolog equivalent to those in procedural languages, the commonly applied code-based testing criteria for conventional software such as statement coverage or branch coverage are not immediately usable for testing Prolog-implemented ESs.

Many research projects are aimed at building software tools to facilitate the construction and validation of ESs. To establish objective quality assurance procedures, formal criteria need to be defined for the testing of ESs so that testing efforts may be measured and directed quantitatively. In this paper, a collection of testing criteria for ESs is defined and analyzed. These criteria are designed specifically for testing the Knowledge Bases (KBs) of ESs that are implemented in logic programming languages.

I Introduction

Expert systems (ESs) have found a great number of applications in various areas. Since the failure of ESs may result in costly losses, establishing quality assurance procedures to ensure their reliability is of great importance to the ESs developers.

It has long been recognized that program testing is an expensive undertaking [Myer79]. It is estimated that up to 50% of the total software development costs goes to testing. Considerable advances on testing methodologies for traditional software have been made during the past two decades.

Many research projects are aimed at building software tools to facilitate the construction and validation of ESs. To establish objective quality assurance procedures, formal criteria need to be defined for the validation of ESs so that testing efforts may be measured and directed quantitatively.

For conventional software, the more widely used code–based testing criteria are statement coverage, branch coverage, and (variations of) path coverage, which require that test data exercise every node, branch, or path, respectively, in the program graph [Myer79]. Another

group of testing criteria that has been studied more recently deals with data–flow testing which is based on exercising selected paths in a program flowgraph with test data [FrWe88, LaKo83]. These criteria were proposed to complement the control–flow–based criteria.

Test criteria for ESs have been proposed in [ViAy90] which encompass both the coverage of elementary entities and the flow paths of a KB system. One criterion—the coverage of elementary entities, specifies the generation of test data that would manipulate all the rules, facts, classes of objects, and their attributes in the KB system. Another criterion relies on the precedence graph of the rules and facts in the KB to cover all the flow paths in the KBS. This scheme closely resembles the branch and path coverage criteria in conventional imperative software testing.

Culbert, *et al.* put forth the fundamental issues of ES validation and verification in [CuRS87]. They pointed out the importance of a well founded methodology of system development in carrying out effective validation and verification processes. General as well as specific issues such as those used for the NASA space program on Validation & Verification of ESs are very well documented in this treatise.

There are a number tools that have been built specifically for ESs validation. One well-known system is the TEIRESIAS program [DaLe82] which was written in order to automate the knowledge base debugging process for MYCIN. Programs for verifying KBs completeness and consistency are discussed in [SuSS82, CrSt87, and PLPN89]. An ongoing project at Lockheed, the EVA (Expert System Validation Associate), is described in [SCSC87]; the goal of the project is to create a generic tool which can validate applications written in any ES shell by translating the language of the shell into a declarative metalanguage.

II Testing Criteria for Expert Systems

Error Classification for ES Software

In [GiRi89], errors in ESs are classified as follows:

◇ *expert knowledge error.* This is an error that would likely be attributed to the expert. The expert may give an erroneous information which is propagated in the system.

◇ *semantic error.* This is an error that is caused by miscommunication between the expert and the knowledge—a situation wherein the expert misunderstands the knowledge engineer, or the expert gives the correct information but the knowledge engineer makes an erroneous translation or entry into the knowledge base, i.e., the knowledge engineer misunderstanding the expert, or vice–versa.

◇ *syntax error.* This is an error that is caused by incorrect forms of rules and/or facts that were entered into the knowledge base.

◇ *inference engine error.* This is an error that is caused by incorrect pattern matchings and resolutions.

Errors are detected during testing by observing that the answer (or conclusion, advice, etc.) produced by the ES on an input query differs from the correct answer (intended by the expert or experts whose knowledge is encoded in the KB); or, for those applications where correctness of reasoning is important, by observing that the sequence of reasonings made by the ES on an input query differs from the correct sequence. In this paper we do not address

the issue of ES evaluation; it is assumed that an oracle capable of deciding the correctness of test results, whether it be a human expert or a group of experts, exists to help conduct testing. We make the further assumption that the expert system shell is correct; and thus we restrict ourselves to testing the KBs of ESs.

Due to the lack of control–flow and data–flow concepts in logic programming equivalent to those in procedural language programming, the commonly applied code–based testing criteria for traditional software, such as statement coverage or branch coverage, are not directly applicable to ESs implemented in logic programming languages. Consequently, testing criteria for ESs need be specifically defined to facilitate objective measurement of testing coverage and to guide testing efforts. In this section, we define and analyze several code–based testing coverage criteria for rule–based ESs written in logic programming languages such as Prolog.

Control–Flow–Based Criteria

We start by defining three control flow testing criteria: clause coverage, path coverage, and branch coverage. These criteria are defined to require the coverage of all nodes, all edges, and all paths in the rule–flow graph of the ES. (A rule–flow graph is a graphical representation of a knowledge base with nodes representing clauses and edges representing clause invocation sequence [ChSt88].)

To overcome the possible deficiencies of control–flow–based criteria, some data–flow–based criteria are also presented. They are defined in terms of variable definitions and uses, and serve as the logic program testing counterparts of data–flow criteria for conventional procedural program testing.

Clause Coverage Criterion

The simplest testing criterion for logic programs is clause coverage, that is, exercise each clause at least once, where a clause is "exercised" when it's head literal is unified with a current goal during a search.

Consider the KB of an ES containing n clauses Q_1, Q_2, ..., Q_n, where each clause has a maximum of m subgoals (a clause with a head and subgoals is a rule, and a clause with a head and no subgoals is a fact), thus clause Q_i has the general form Q_i :- $Q_{i1}, Q_{i2}, ..., Q_{im}$. Suppose that Q_i has arity k, then Q_i can be invoked by a query ?-$Q_i(X_1, X_2, ..., X_k)$, where all the X_i's are variables. Thus, testing an ES with n clauses using clause coverage requires a maximum of $O(n)$ queries or test cases.

Path Coverage Criterion

A most thorough testing of a KB would involve testing all possible input queries or all feasible searches. A search can be represented by a path, namely a sequence of numbers indicating the clauses that are selected for unification during the search. Since the set of all feasible searches can easily be infinite, this criterion of path coverage—an essentially similar criterion to the path coverage of conventional software testing—is not practical.

To approximate path coverage, we can make a partition of the set of feasible paths into a finite number of subsets, and then select a representative path from each subset. Finally,

test cases or queries are constructed to traverse the selected representative paths. However, since it is generally impossible to decide whether an arbitrarily given path is feasible, the set of all feasible paths can not be known beforehand. To implement a path coverage testing method, rule–flow graphs can be used to describe all paths (including infeasible ones).

For example, figure 1 is a rule-based system and its rule–flow graph, taken from [ChSt88]. The set of all paths of this system can be described by the regular expression

13*25 + 13*26 + 13*27 + 13*28 + 14*25 + 14*26 + 14*27 + 14*28 + 1(3+4)*25 + 1(3+4)*26 + 1(3 + 4)*27 + 1(3 + 4)*28.

There are 12 terms in this expression, each of which may be considered an "equivalence class" in that different paths which are described by the same term differ only in the number of times a particular loop—indicated by *, the closure operator—is traversed. So, if two representative paths are selected from each term (say, one of them skips the loop and one of them traverses the loop once), a set of sixteen paths is obtained, e.g.,

{125, 126, 127, 128, 1325, 1326, 1327, 1328, 1425, 1426, 1427, 1428, 13425, 13426, 13427, 13428}.

For each path in the set, a query is then designed to traverse it. In this case, four test cases are generated since only four of the sixteen paths—128, 1325, 1428, 13428—are feasible, i.e. these are the only paths that can possibly be traversed in the program.

(1) grade(Name, Grade) :- student(Name, Answers),
expect(N_questions, Correct_answers),
r_answers(Answers, Correct_answers, N_rights),
Ratio is N_rights/N_questions,
Score is Ratio*100,
compute_grade(Score, Grade).

(2) r_answers([ ],[ ],0).

(3) r_answers([X|Y],[X|Z],R1) :- r_answers(Y, Z, R),
R1 is R + 1.

(4) r_answers([_|Y],[_|Z],R) :- r_answers(Y, Z, R).

(5) compute_grade(Score,a) :- Score > = 90.

(6) compute_grade(Score,b) :- Score < 90, Score > = 80.

(7) compute_grade(Score,c) :- Score < 80, Score > = 70.

(8) compute_grade(Score,f) :- Score < 70.

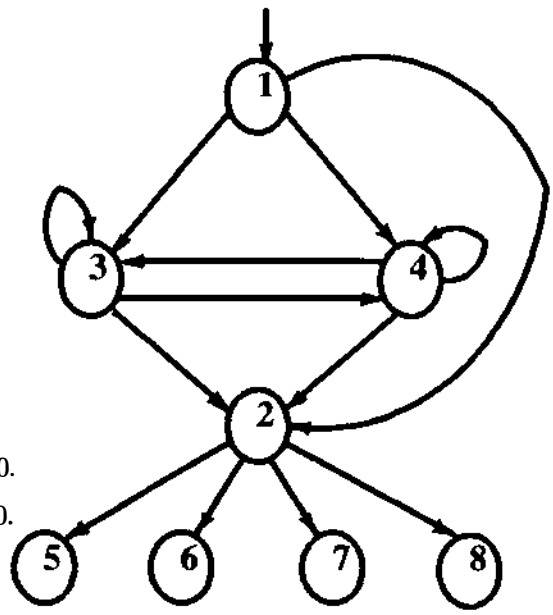


Figure 1 : ES 1 and its Rule–Flow Diagram

Branch Coverage Criterion

Branch coverage testing—to cover all the branches of the flowchart of a program [Myer79]—has often been applied by industry as one of the quality assurance steps for conventional software. For ESs, a similar branch coverage criterion can be defined in terms of rule–flow graphs, that is, generate enough test cases to cover all the branches of the rule–flow graph. This is almost the same as the *node coverage* criterion as given in [ChSt88], except that the goal here is to cover all branches rather than all nodes. We note that the node coverage criterion of [ChSt88] is essentially the "statement coverage" criterion for Prolog programs.

For example, to do a branch coverage testing of the system given in figure 1, a minimum of four test queries need be designed to cover all the branches in the graph, since there are four directed edges from node 2 to the four terminal nodes. The set of four paths $\{128, 1325, 14333326, 1333334427\}$ is a minimal test set to satisfy the branch coverage criterion because all the four paths are feasible and they cover all the branches in the graph. Branch testing requires no more than $O(n)$ test queries.

Branch testing is not able to uncover many kinds of simple errors. In the above example, the four test cases cover all the branches and all the clauses in the KB; however, if the predicate "$\geq$" in the subgoals of clauses (5) or (7) were erroneously written as ">", then the error will not be detected by branch coverage. A stronger criterion than branch coverage would require that, in addition to all the branches, all the conditions take on all possible outcomes at least once. We call this criterion "branch condition coverage", which is equivalent to the "multiple condition coverage" criterion in conventional software testing [Myer79].

In the previous example, to do branch condition coverage, sufficient test queries must be generated to cover the seven conditions Score is <70, =70, 70 < Score <80, =80, 80 < Score <90, =90, >90. This would require some longer input queries, for instance, to make the Score exactly equals to 70 or 90 requires that the shortest queries contain ten answers, or a multiple of ten.

Since in many KBs a large number of the rules are written to merely encode "IF_THEN_ELSE"-type structure in decision making, a testing procedure that applies branch condition coverage to all such rules will be more useful than branch coverage in detecting errors. It can be estimated that branch condition coverage requires $O(n * m)$ test cases.

Data–Flow–Based Criteria

The following definitions of data–flow–based criteria have been greatly influenced by the work of Rapps and Weyuker [RaWe85] on the same subject for traditional software.

We will start with some basic definitions. A *variable definition* is the first appearance of a variable in a predicate (we assume that every variable name is distinct in the program; furthermore, a variable must not be an alias of a variable that has been previously defined). A variable can be used for either computation or in a boolean predicate.

A *start* node of a rule–flow graph corresponds to a clause that is invoked from the topmost query. An *exit* node is a leaf node in the rule–flow graph. A *path*, P, is a finite sequence of nodes in a rule–flow graph that commences at a start node and ends at an exit node.

def(i) is the set of variables that are defined in node i; *c–use(i)* is the set of variables that are used for computation in node i; *p–use(i)* is the set of variables used in boolean predicates in node i.

Following some of the concepts first introduced in [RaWe85], we define three data–flow criteria.

All–du–paths Criterion

A *du–path wrt X*, where X is a program variable, is a path that includes a pair of nodes i and j, such that $X \in def(i) \cap c\text{–}use(j)$ or $X \in def(i) \cap p\text{–}use(j)$. A set of paths, D, satisfies the all–du–paths criterion if it includes all du–paths of all variables.

All–dc–paths Criterion

A *dc–path wrt X*, where X is a program variable, is a path that includes a pair of nodes i and j, such that $X \in def(i) \cap c\text{–}use(j)$. A set of paths, D, satisfies the all–dc–paths criterion if it includes all dc–paths of all variables.

All–dp–paths Criterion

A *dp–path wrt X*, where X is a program variable, is a path that includes a pair of nodes i and j, such that $X \in def(i) \cap p\text{–}use(j)$. A set of paths, D, satisfies the all–dp–paths criterion if it includes all dp–paths of all variables.

The all–du–paths criterion requires the coverage of all the usage in computations and in boolean predicates of all variables; the all–dc–paths criterion (all–dp–paths criterion), requires that all computation uses (all boolean predicate uses) of a variable be covered.

Given n clauses in an ES, each of the three data–flow testing criteria requires a maximum of $O(2^n)$ queries to be satisfied, even though this worst case scenario should be extremely unlikely (most ESs should need no more than $O(n^2)$ test cases to satisfy the three criteria).

As an example, consider the following program and query:

---

(1) $Q_1(A, B, AB) :- P_1(A, AB),\ P_2(A, B),\ P_3(B, D).$
(2) $P_1(A, C) :- A\ is\ 2 * C.$
(3) $P_2(A, B) :- A =< 0,\ B\ is\ ABS(A).$
(4) $P_2(A, B) :- A > 0,\ B\ is\ SQRT(A).$
(5) $P_3(B, D) :- D\ is\ B + 12.$

$:- Q_1(A, B, 5, X).$

Two dp–paths for variable A are 1–2–3–5 and 1–2–4–5, note that these two also happen to be dc–paths wrt A; two dc–paths for variable B are 1–2–3–5 and 1–2–4–5; To satisfy the all–du–paths criterion wrt variable A, both paths 1–2–3–5 and 1–2–4–5 must be exercised.

Subsumption Relations of Criteria

A testing coverage criterion $C1$ *subsumes* another criterion $C2$, denoted $C1 \rightarrow C2$, if any set of test data that satisfies $C1$ also satisfies $C2$. If neither one subsumes the other, the two criteria are said to be *incomparable* [DaWe83]. The subsumption relations among the families of data–flow and control–flow criteria is as follows: path coverage → all–du–paths → branch coverage → node coverage → clause coverage; and all–du–paths → all–dc–paths; and all–du–paths → all–dp–paths.

Dynamic and Incremental Criteria

The previous criteria apply to prolog programs that do not have the ability to alter themselves at runtime. Unfortunately, for some ESs, this is not the case since the dynamic behavior—however controversial—may be required for knowledge elicitation or for other application–specific reasons. This self-altering effect occurs whenever modifying predicates, such as *assert* and *retract* in Prolog, *build* and *remove* in OPS5 [BFKM85], and *excise*, *assert*, and *retract* in CLIPS [Giar89], among others, are invoked.

We first give the definitions of some terms, some of which are adapted from the logic program flow analysis work of Debray [Debr87]. A predicate p is *dependent* on a predicate q in a program, written $p\mathrm{D}q$, if q occurs in a body of a clause whose head unifies with p. An example clause is $p \ :- \ \ldots \ q \ \ldots$. A predicate q is *reachable* from predicate p, written $p\mathrm{R}q$, if either $p\mathrm{D}q$ or there exists a predicate r such that $p\mathrm{D}r$ and $r\mathrm{R}q$. Thus, R is a transitive relation. The *reachable set* of p, denoted $T(p)$, contains p itself and all predicates reachable from it, i.e., $T(p) = \{q \mid p\mathrm{R}^*q\}$, where R^* is the reflexive and transitive closure of R. The system predicates *assert* and *retract*, and predicates containing *assert* or *retract* as subgoals are called *modifying predicates*. Suppose p is a modifying predicate. $G(p)$, the *globally reachable set* wrt to p, is a subset of $T(p)$; and it contains non-modifying predicates in $T(p)$ that are reachable from predicates not in $T(p)$. $I(p)$, the *independent set* wrt p, is $T(p) - G(p)$.

Example

Given the following program

$$a \ :- \ q, \ s.$$
$$p \ :- \ q, \ assert((r \ :- \ t, \ u)), \ s.$$
$$m \ :- \ retract((n \ :- \ y, \ z)).$$

In addition to *assert* and *retract*, p and m are modifying predicates; $T(p) = \{p, q, r, s, t, u\}$ and $T(m) = \{m, n, y, z\}$ are reachable sets of p and m; $G(p) = \{q, s\}$ and $G(m) = \emptyset$ are globally reachable sets; $I(p) = \{p, r, t, u\}$ and $I(m) = \{m, n, y, z\}$ are independent sets.

Dynamic Data–Flow Criteria

Let $T(a)$ ($T(r)$) be the reachable set wrt a, an assertive predicate (r, a retractive predicate) and D be a set of test data.

D satisfies the *all–asserts* criterion if for every set $T(a)$, in the system, where a is assertive, there exist test data from D that cause every predicate in $T(a)$ to be invoked at least once. This criterion requires that all the predicates that are reachable from a modifying predicate

be tested; stated differently, all the parts in the program that can be reached by the ripple effect of an added clause need to be tested. It augments the static clause coverage criterion since it requires that all clauses, which are not present during the static analysis phase and are added during runtime, are to be exercised at least once.

D satisfies the *all–retracts* criterion if for every set $T(r)$ in the system, where r is retractive, there exist test data that cause(s) every predicate in the set $G(r)$ to be invoked at least once. This criterion ensures that predicates that are reachable from both r (the retractive predicate) and other predicates in the program are exercised. Furthermore, it only requires coverage of globally reachable sets because after the retractive predicate r is invoked, the independent sets contain no reachable predicates.

Incremental Validation Criteria

Even as the aforementioned criteria are meant to address the static and dynamic nature of the system, the inherent characteristics of ESs—knowledge expansion and contraction, is another area that needs to be also considered. As rules and facts are added to and deleted from the system, re-establishing confidence in it through regression testing is required. Thus, methods and criteria to facilitate regression testing need also be developed.

An effective regression testing (or incremental validation) method should be involved only on parts of the system that are affected by the expansion or contraction of its rules and facts. Thus, we need to define criteria that will concentrate on these local changes.

In the following, we introduce incremental validation criteria based on two categories: rule–based and data–based. The rule–based criteria concern the consistency and completeness of the ES as it evolves. The data–based criteria will depend on incremental data–flow analyses of the system after its modification.

Two rule–based validation criteria are the consistency criterion and the completeness criterion. The *consistency* criterion is satisfied whenever a new rule is entered into the system, the system is checked for consistency; for example, a consistency algorithm may check for (a) redundancy, (b) conflict, (c) subsumption, (d) unnecessary clauses, and (e) circularity. The *completeness* criterion is satisfied whenever a new rule is entered into the system, the system is checked for completeness; for example, a completeness algorithm may check for reachability and attribute validity [NPLP85]. Note that these two criteria are not testing criteria unless the consistency and completeness checking algorithms actually involve testing the ES on selected queries.

Two data–based incremental testing criteria are the all–new–paths criterion and the all–modified–paths criterion. The *all–new–paths* criterion requires testing all the new du–paths of all variables after an ES is modified. The *all–modified–paths* criterion requires testing all du–paths of all variables that have been modified after an ES is modified. (Let S1 be the set of all du–paths of the original ES, and let S2 be the set of all du–paths of the new ES. Then S2 – S1 is the set of all new du–paths; and S1 $\cap$ S2 gives the set of all modified du–paths.)

Some Functional Criteria

Similar to structural or code-based testing, the basic idea of functional testing is coverage—generate enough test cases to exercise all the functional components of the ES, and a usual method to achieve coverage is partitioning—the input domain is partitioned into (not necessarily disjoint) sub-domains containing inputs which cause the ES to behave in

similar manners, and then a test set is formed by selecting a representative input from each sub-domain.

Functional testing criteria and test generation methods such as equivalence partitioning, special value testing, and cause-effect graphing are general enough to be applicable to ESs. Since it is common for ESs to have finite output space with discrete outputs (e.g., identification systems), the method of output equivalence partitioning is particularly useful.

For example, the ES for grading (figure 1) produces four different answers—the grades of a, b, c, and f. To conduct a test based on output equivalence partitioning, a test set containing four cases need be generated to cause the ES to give the four different answers. Phrased as a criterion, this amounts to exercise enough test cases to produce each different answer or conclusion at least once. This criterion clearly requires only a number of test cases equal to the number of conclusions in the ES. (Simpler identification systems may even have finite discrete input space. Such systems essentially implement decision tables or boolean functions and can be more easily tested, see the next section).

Two other functional testing criteria for ESs are to invoke all the questions that can be asked by the ES of the user, and to invoke all possible sequence of reasonings. The former criterion is important for any ES that interactively solicits information from users, the latter criterion is important for ESs whose correctness depend not only on giving correct conclusions but also on making the correct sequence of reasoning.

To apply the criterion of covering all sequences of reasoning, a set C = $\{C_1, C_2, ..., C_k\}$, containing all intermediate and final conclusions in the ES that are critical to correctness is first identified. Then test cases must be selected to exercise all sequences of reasoning $< C_{i1}, C_{i2}, ..., C_{id} >$ that can possibly be made by the ES, where $C_{ij} \in \{C_1, C_2, ..., C_k\}$ is the j^{th} intermediate conclusion in the sequence of reasoning. In the worst case the number of test cases required becomes exponential in d, the length of a reasoning sequence of maximal length, although this should be extremely unlikely for most ESs (k*d is a more realistic estimate of the upper bound on the number of test cases required). The set C should be judiciously selected to contain only the critical intermediate and final conclusions of the ES such that there is only a finite number of possible reasoning sequences. Therefore, the (potentially infinite) input domain is mapped to the finite domain of all possible reasoning sequences, and the criterion provides another approximation of path testing.

A deficiency of rule-flow diagrams is that they only indicate the possible sequence of rule invocations during a search but can not represent the backtrackings that may occur during a search for multiple solutions. Clauses with the same head in a KB can be classified as mutually exclusive—those that will never be invoked together for any input query, such as clauses (5), (6), (7), (8) of example 1; mutually inclusive—those that will always be invoked together during a search, such as clauses (3) and (2), (4) and (2) of example 1, or clauses (3) and (4), (5) and (6), (7) and (8) of ES 2 (figure 2); or mutually non-exclusive—those that may or may not be invoked together for an input query, such as clauses (3) and (4) of example 1. Correspondingly, testing criteria can be defined and analyzed for different levels of coverage of the three classes of clauses. For example, testing can be conducted against the criterion that all sets of mutually inclusive clauses must be covered by test inputs, or under the criterion that all sets of mutually non-exclusive clauses must be covered by enough test inputs that demonstrate both inclusion and exclusion.

The latter criterion, when applied to ES 1, requires at least two test inputs—one containing all correct (or all incorrect) answers to show exclusion of clauses (3) and (4);

the other containing both correct and incorrect answers to show inclusion of (3) and (4)—since (3) and (4) are mutually non-exclusive. For ES 1, this test set of two inputs also happen to satisfy the criterion of covering all sets of mutually inclusive clauses, since there are only two sets of mutually inclusive clauses, namely ((3), (2)) and ((4), (2)).

```
[1]  facility(Pers,Fac) :- book_due(pers,Book),!, basic_facility(Fac).
[2]  facility(Pers,Fac) :- gen_facility(Fac).
[3]  basic_facility(reference).
[4]  basic_facility(enquiries).
[5]  additional_facility(borrowing).
[6]  additional_facility(inter_library_loan).
[7]  gen_facility(X) :- basic_facility(X).
[8]  gen_facility(X) :- additional_facility(X).
[9]  book_due("C. Walter",book1009).
[10] book_due("G. Brown",book6512).
[11] book_due("M. York",book3355).
```

Figure 2: ES 2 (A library facility program) [ClMe84]

Mutation analysis is an error–based testing method that is directly applicable to ESs software. One simple way to define a mutant operator is to interchange the conclusions of the ES, i.e., mutants are obtained by replacing every conclusion with every other conclusion. For an ES containing n conclusions, this gives a set of $n^2 - n$ mutants; then test data are generated to kill the non-equivalent mutants.

Consider ES 1, for example, four different conclusions are produced by the ES, so twelve mutants are generated by the mutant operator. If the ES under test is correct, then all twelve mutants are incorrect and non-equivalent, a set of four test queries, where each query leads the ES under test to a different conclusion, will be sufficient to kill off all twelve mutants. It is easy to see that for small ESs like ES 1, a test set that is generated for mutation analysis or by output equivalence partitioning will also satisfy (or close to satisfy) the conclusion coverage criterion.

III Testing Procedure and Test Data Generation

A Testing Procedure for Expert Systems

How thoroughly does an ES need to be tested? The answer, of course, depends on the application served by the ES and its reliability requirements; 90% branch coverage might be good enough for most ESs while totally unacceptable for others. The testing coverage criteria presented in the preceding sections provide a basis for measuring testing thoroughness of ESs. A feasible testing procedure that is reasonably inexpensive and useful is recommended as follows:

- ◇ step 1: Check for consistency and completeness of the ES. (There are a number of algorithms in the literature for doing this, e.g. [NPLP85, CrSt87, Puur87]);
- ◇ step 2: Select a functional criterion, and generate test cases to satisfy the criterion. Test the ES on these cases.

- step 3: Select a structural criterion that is incomparable to, or that subsumes the functional criterion used in step 2, and generate more test cases to satisfy the criterion. Test the ES on these supplemental test cases.

In steps 2 and 3, test cases may be generated incrementally.

Referring to ES 1 (fig. 1), after checking for consistency and completeness, we select "covering all conclusions" as the functional criterion, and generate the following four test cases to exercise the four conclusions (grades of a, b, c, and f, or clauses 5, 6, 7, and 8):

test case 1:	input: (c c c)	conclusion: grade a
test case 2:	input: (c c c c i)	conclusion: grade b
test case 3:	input: (c c c i)	conclusion: grade c
test case 4:	input: (c i)	conclusion: grade f

A "c" represents a correct answer and an "i" represents an incorrect answer in the above input cases. After testing the ES on the four cases and seeing correct results, we proceed to step 3 and choose, say, "branch coverage" as the functional criterion. Since the four test cases collectively have covered all the edges in the rule-flow diagram except edges (1,4), (4,3), and (4,4), test case 5 is derived to supplement test cases 1–4:

test case 5:	input (i i c)	conclusion: grade f

After testing the ES on test case 5 and seeing correct results, 100% branch coverage is achieved and the testing is completed.

Test Data Generation for Simple ESs

In this section, two models of ES rule representations are presented—as boolean expressions and as a digital circuit. We illustrate that test data generation methods for fault detection in either representation are directly applicable to the generation of test data for simple ESs that are implemented exclusively with atomic conditions.

A rule–based ES comprises three basic components: an inference engine, a KB, and a user interface. The KB contains a set of rules, each rule consists of a premise and a conclusion. The action on the conclusion is carried out once the premise is determined to be true. Examples of rules extracted from a hypothetical EENT (Eye–Ear–Nose–Throat) diagnostic ES (diagnostic premises were obtained from [Berk77]) are the following:

```
RULE 1:
IF otorhino_group AND NOT ear_discharge AND fever AND ear_passage_clear
THEN mastoiditis

RULE 2:
IF otorhino_group AND NOT ear_discharge AND NOT fever
AND NOT ear_passage_clear THEN ear_obstruction

RULE 3:
IF opthalmologic_group AND eye_discharge AND NOT fever
AND eyelid_swelling THEN conjunctivitis
```

```
RULE 4:
IF opthalmologic_group AND NOT eye_discharge AND blurred_vision
THEN glaucoma
```

A rule–based ES is in *canonical form* if every implication operator is replaced by a disjunction operator and every premise clause is negated. A close look at the rules of a simple ES in canonical form reveals that they are equivalent to boolean expressions. For example, the rules of the above ES can be represented by the following boolean expressions:

RULE 1: ¬(Group_1 & ¬ Ea_1 & F_1 & Ea_2) ∨ Ma

RULE 2: ¬(Group_1 & ¬ Ea_1 & ¬ F_1 & ¬ Ea_2) ∨ Eo

RULE 3: ¬(Group_2 & Ey_1 & ¬ F_1 & E_2) ∨ Co

RULE 4: ¬(Group_2 & ¬ Ey_1 & Ey_3) ∨ Gl

The EENT Expert System Rules as Boolean Expressions

The only boolean operator that does not appear in the above is the disjunction operator OR which will be represented by the symbol '+'. Thus a rule such as

```
IF (Cond_1 OR Cond_2) AND NOT Cond_3
THEN Conclusion_X
```

will be translated into a boolean expression of the form

¬((C_1 + C_2) & ¬ C_3) ∨ C_x

Since there is a clear isomorphism between a canonical from rule–based ES and a set of boolean expressions, a test data generation algorithm from [TaSu87] for boolean expressions can be used to generate test data for ESs.

Test Data Generation for Boolean Expressions

The test data generation algorithm is based on attribute grammars [AhSU86]. The attribute grammar approach uses a predefined context–free grammar where each production has a corresponding attribute rule associated with it. A general description of the test data generation algorithm is as follows: a boolean expression, B, is assigned two attributes, T(B) and F(B) such that

(i) T(B) ∪ F(B) is the test set produced for B,
(ii) For each test X in T(B), B(X) is true, and
(iii) For each test Y in F(B), B(Y) is false.

To generate a test set using the above description, a bottom–up parsing of B based on the production rules of G is made. Upon each reduction, an appropriate attribute rule is applied. This continues until B is completely parsed. The result of this process is the test set T(B) ∪ F(B). For a detailed description of the algorithm, refer to [TaSu87].

An Example

Given RULE 1 as described above, the test data generator proceeds to find the test sets as follows:

(i) calculate the test data for (F_1 & Ea_2). This should yield the set $\{(t,t),(t,f),(f,t)\}$,

(ii) calculate the test data for ($\neg$ Ea_1). The set for this is $\{(t),(f)\}$,

(iii) calculate the test data for (Group_1 & $\neg$ Ea_1). The result for this should be $\{(t,f),(f,f),(t,t)\}$,

(iv) finally, the calculation for the whole boolean expression corresponding to RULE 1 (Group_1 & $\neg$ Ea_1 & F_1 & Ea_2) yields the set $\{(t,f,t,t),(f,f,t,t),(t,t,t,t),(t,f,f,t),(t,f,t,f)\}$.

The advantage of using the above algorithm is the guaranteed detection of boolean operator errors using at most $n+1$ test sets for n variables. It has been proved in [TaSu87] that the minimum number of test set, $|T_{min}|$, for n variables, satisfies the equation $2\sqrt{n} \leq |T_{min}| \leq n+1$. The same result holds for the class of completely fan–out free digital circuits [Haye71].

Test Data Generation from Digital Circuit Models

Digital circuit testing methodologies are very well established [Fuji85]. By modeling the KB of a simple ES as a digital circuit, methods for testing digital circuits can be used to test ESs rules.

The most widely used fault model for digital circuits is the **stuck-at-fault model** [Koha78]. This model is the basis of many test data generation methods for boolean expressions; it also inspired the mutation analysis method for software [Budd81].

Here we give an example of test data generation for a simple ES using a digital circuit testing method. Consider RULE 1 and its equivalent 2–level digital circuit containing three AND gates (implementing $f = ((A \cdot \bar{B}) \cdot (C \cdot D))$). The fault table method of test pattern generation [Koha78] will be used to generate the tests. The minimal set of fault–detection test patterns contains five entries: (0011,A_1), (1001,C_1), (1010,D_1), (1011,$A_0B_1C_0D_0$), and (1111,B_0), the first tuple gives the test pattern 0011 for the A–stuck–at–1 fault, etc.

IV Conclusion

We have defined and analyzed several simple testing coverage criteria for ESs that are implemented in a logic programming language like Prolog. These criteria can be used for testing ordinary Prolog programs as well.

Although the structure of logic programs is very different from that of programs written in conventional procedural languages, the criteria defined in this paper are much similar to the testing criteria for conventional software. Unfortunately, the undecidable and computationally difficult problems related to conventional program testing also carry over to the testing of logic programs and ESs.

For example, given a program and a statement (branch, path) of it, it is not decidable whether this statement (branch, path) is feasible. For logic programs, it is similarly undecidable whether a clause is feasible (i.e., whether there exists an input query that will

cause a given clause within a given program to be invoked during execution is undecidable), or whether a given branch or path is feasible. Since many of the code-based testing criteria are usually not satisfiable for large ESs, percentage goals should be set for monitoring testing coverage, e.g., 90% branch coverage or 95% clause coverage.

It is believed that neither functional testing nor structural testing will be very effective alone; therefore, software testing procedures normally include both functional testing and structure-based testing. For ESs software, the KB also needs to be checked for consistency, completeness, and to detect defective rules—unreachable rules, redundant rules, subsumed rules, etc. One main advantage of applying a formal criterion as a quality assurance step is that a quantitative measurement of testing coverage can be made.

The testing and validation problem for ESs will become increasingly important as the applications of ESs become more widespread. We will continue to pursue this work and hope that more research efforts will be focused on this topic to facilitate the development of better ESs technologies.

V References

[AhSU86] Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers, Principles, Techniques, and Tools*, Addison Wesley, 2nd Edition, 1986.

[Berk77] Berkow, Robert, *The Merck Manual of Diagnosis and Therapy*, Merck Sharp & Co., Inc., 13th Edition, 1977, pp. 1618–1649.

[BFKM85] Brownston, L., Farrel, R., Kant, E., and Martin, N., *Production Expert Systems in OPS5 An Introduction to Rule-Based Programming*, Addison Wesley, Reading, MA, 1985.

[Budd81] Budd, T.A., "Mutation Analysis: Ideas, Examples, Problems, and Prospects," *Computer Program Testing*, Chandrasekaran, B. and Radicchi, S., eds., North-Holland, Amsterdam, pp. 129–148.

[ChSt88] Chang, C.L. and Stachowitz, R.A., "Testing Expert Systems," Proceedings of the Space Operations Automation and Robotics (SOAR 88) Workshop, Dayton, Ohio, July, 1988, pp. 131–135.

[Chap82] Chapman, David, "A Program Testing Assistant," *Communications of the ACM*, Vol. 25, no. 9, Sept. 1982, pp. 625–634.

[Chus83] Chusho, Takeshi, "Coverage Measure for Path Testing Based on the Concept of Essential Branches," *Journal of Information Processing*, Vol. 6, no. 4, 1983, pp. 199–205.

[Clar76] Clarke, Lori, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Trans. on Software Engg.* SE-2, no. 3, 1976, pp. 215–222.

[ClMe84] Clocksin, W.F. and Mellish, C.S., *Programming in Prolog*, 2nd Ed., Springer-Verlag, Berlin, 1984.

[CoHu87] Cochran, E.L. and Hutchins, B.L., "Testing, Verifying, and Releasing an Expert System: The Case History of Mentor," Proceedings of the 3rd International Conference on Artificial Intelligence Applications, Kissimmee, Florida, 1987, pp.163–167.

[CrSt87] Cragun, B.J. and Steudel, H.J., "A Decision-Table Based Processor for Checking Completeness and Consistency in Rule-Based Expert Systems," in *International Journal of Man-Machine Studies*, Vol. 26, 1987, pp. 633–648.

[CuRS87] Culbert, C., Riley, G., and Savely, R.T., "Approaches to the Verification of Rule–Based Expert Systems," *First Annual Workshop on Space Operations Automation and Robotics*, NASA Conf Publication 2491, Houston, TX, August, 1987, pp. 191–196.

[DaLe82] Davis, R. and Lenat, D.B., *Knowledge-Based Systems in Artificial Intelligence*, McGraw–Hill, NY, 1982.

[DaWe83] Davis, M. and Weyuker, E.J., "A Formal Notion of Program–Based Test Data Adequacy," *Information and Control*, Vol. 56, no. 1–2, January, 1983, pp. 51–72.

[FrWe88] Frankl, P.G. and Weyuker, E.J.,"An Applicable Family of Data Flow Testing Criteria," *IEEE Trans. on Software Engg.*, Vol. 14, no. 10, October, 1988, pp. 1483–1498.

[Fuji85] Fujiwara, H., *Logical Testing and Design for Testability*, MIT Press, Cambridge, MA, 1985.

[Giar89] Giarratano, Joseph C., *CLIPS User's Guide* (Version 4.3), Artificial Intelligence Section, Johnson Space Center, October 1989.

[Haye71] Hayes, John P., "On Realizations of Boolean Functions Requiring a Minimal or Near–Minimal Number of Tests," *IEEE Trans. on Computers*, Vol. C–20, No. 12, December 1971, pp. 1506–1513.

[Howd75] Howden, William E., "Methodology of the Generation of Program Test Data," *IEEE Trans. on Computers*, Vol. C–24, No. 5, May 1975, pp. 554–559.

[Howd76] Howden, William E., "Reliability of the Path Analysis Strategy," *IEEE Trans. on Software Engg.*, Vol. SE–2, No. 3, 1976, pp. 208–215.

[Huan75] Huang, J.C., "An Approach to Program Testing," *ACM Computing Surveys*, Vol. 7, No. 3, Sept. 1975, pp. 113–128.

[Ince87] Ince, D.C., "The Automatic Generation of Test Data," *The Computer Journal* Vol. 30, no. 1, 1987, pp. 63–69.

[Koha78] Kohavi, Z., *Switching and Finite Automata Theory* McGraw–Hill, N.Y., 1978.

[Kore90] Korel, Bogdan, "Automated Software Test Data Generation," *IEEE Trans. on Software Engg.* Vol. 16, no. 8, August 1990, pp. 870–879.

[LaKo83] Laski, J.W. and Korel, B., "A Data Flow Oriented Program Testing Strategy," *IEEE Trans. on Software Engg.*, Vol. SE–9, May 1983, pp. 347–354.

[MaRy90] Marlowe, T.J. and Ryder, B.G.,"An Efficient Hybrid Algorithm for Incremental Data Flow Analysis," Proc. of the 1990 ACM Conference on Principles of Programming Languages, ACM Press, 1990, pp. 184–196.

[Myer79] Myers, Glenford J., *The Art of Software Testing*, John Wiley and Sons, N.Y., 1979.

[NPLP85] Nguyen, T.A., Perkins, W.A., Laffey, T.J., and Pecora, D., "Checking an Expert Systems Knowledge Base for Consistency and Completeness," Proc. of the 9th IJCAI Conf., Los Angeles, CA, 1985, pp. 375–378.

[Ntaf84] Ntafos, Simeon C., "On Required Element Testing," *IEEE Trans. on Software Engg.*, Vol. SE–10, no. 6, 1984, pp. 795–803.

[OLea87] O'Leary, D.E., "Validation of Expert Systems—With Applications to Auditing and Accounting Expert Systems," *Decision Sciences Journal*, Vol. 18, no. 3, 1987, pp. 468–486.

[PLPN89] Perkins, W.A., Laffey, T.J., Pecora, D., and Nguyen, T.A., "Knowledge Base Verification," *Topics in Expert System Design*, Elsevier Science Publishers, North Holland, 1989. pp. 353–376.

[Puur87] Puuronen Seppo, "A Tabular Rule–Checking Method," Proc. of the 7th Int'l Workshop on Expert Systems and Their Applications, Vol. I, Avignon, France, May 13–15, 1987, EC2 Publishing, France, 1987, pp. 257–268.

[RaWe85] Rapps, S. and Weyuker, E.J., "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. on Software Engg.* Vol. SE–11, no. 4, April 1985, pp. 367–375.

[RGOM89] Radwan, A. E., Goul, M., O'Leary, T.J., and Moffitt, K.E., "A Verification Approach for Knowledge–Based Systems," *Transportation Research Journal*, Vol. 23A, no.4, 1989, pp. 287–300.

[RyPa88] Ryder, B.G. and Paull, M.C., "Incremental Data Flow Analysis Algorithms," *ACM Transactions on Programming Languages and Systems*, Vol. 10, no.1, Jan. 1988, pp. 1–50.

[SCSC87] Stachowitz, R.A., Chang, C.L., Stock, J.B., and Combs, J.B., "Building Validation Tools for Knowledge–Based Systems," *First Annual Workshop on Space Operations Automation and Robotics*, NASA Conf Publication 2491, Houston, TX, August, 1987, pp. 209–215.

[Smit88] Smith, Peter, *Expert System Development in Prolog and Turbo–Prolog*, Halstead Press, N.Y., 1988.

[SuSS82] Suwa, M., Scott, A.C., and Shortliffe, E.H., "An Approach to Verifying Completeness and Consistency in a Rule–Based Expert System," *The AI Magazine*, Fall 1982, pp. 16–21.

[TaSu87] Tai, K.C. and Su, H.K., "Test Generation for Boolean Expressions," Proceedings of the IEEE 11th Conf. on Computer Software and Applications, Tokyo, Japan, Oct. 1987, pp. 278–284.

[ViAy90] Vignollet, L. and Ayel, M., "Conceptual Model for Building Sets of Test Samples for Knowledge Bases." Unpublished report.

[WeOs80] Weyuker, E.J. and Ostrand, T.J., "Theories of Program Testing and the Application of Revealing Subdomains," *IEEE Trans. on Software Engg.* Vol. SE–6, May 1980, pp. 236–246.

[WhCo80] White, L.J. and Cohen, E.I., "A Domain Strategy for Computer Program Testing," *IEEE Trans. on Software Engg.* Vol. SE–6, May 1980, pp. 247–257.

[Zade84] Zadeck, F.K., "Incremental Data Flow Analysis in a Structured Program Editor," Proc. of the ACM SIGPLAN Symposium on Compiler Construction, June 1984, *SIGPLAN Notices*, Vol. 19, no. 6, pp. 132–143.

MANAGING IN THE CLEANROOM ENVIRONMENT

Prepared for
9th Pacific Northwest Software Quality Conference
Portland, Oregon
October 7, 1991

Michael Dyer
IBM Federal Sector Division
Bethesda, Md. 20817

THE CLEANROOM METHOD

Cleanroom is the name of a software development method **[1]** which was organized to support the measurement and certification of software Mean-Time-To-Failure (MTTF), prior to the release of software to its user. Cleanroom is also the label for a collection of software engineering methods which are the components of the Cleanroom software development method. The term Cleanroom was selected to draw attention to a development process which strives to prevent the introduction of errors during software development.

The Cleanroom software development process is organized as a set of component methods, which can be applied individually but in combination represent a radical departure from current software development practice. The Cleanroom process extends beyond the boundaries of what is normally interpreted as software development and deals with software specification at one extreme and with functional software testing at the other extreme. Cleanroom introduces new controls for software development, imposes new roles and responsibilities on the various engineering disciplines, eliminates some seemingly core methods from the development process and raises the level of training and proficiency required of the engineering disciplines.

The total Cleanroom process should be used for software development to realize its full potential for enhancing product quality and process productivity. However, transitioning to a totally different development process is not always practical within an ongoing software development environment and an incremental introduction of the Cleanroom components has proven to be a more effective strategy for technology transfer. Each of the half dozen components addresses a specific aspect of the software development process, makes a separate contribution to the development and has a unique set of considerations for process insertion. The components have been used individually and in combination with demonstrable positive results. This incremental realization of positive results generally leads to the gradual introduction of the total process, which can now be accomplished without the trauma of switching to a radically new development process.

Michael Dyer
IBM Federal Sector Division
Bethesda, Md. 20817

The Cleanroom components are organized along the six technical lines of software specification, software development, software correctness verification, independent software product testing, software reliability measurement and statistical process control.

Software Specification

With the Cleanroom process, there is an implied requirement for correctness, completeness and stability in the software specifications, so that the correctness of the software design can be verified as it is elaborated. Cleanroom forces software design against the early specification of requirements and, in that process, forces stability and completeness in these specifications. The result is stricter accountability between specifiers and developers and the early introduction of a controlled approach to stabilizing the product requirements. In the Cleanroom method, more formal notation is introduced for accuracy and to resolve many of the issues which would be subsequently raised by the software designer, attempting to verify the correctness of a design. The specification content is broadened to identify the packaging of software requirements into incremental releases and to establish the reliability (MTTF) targets for the product. Cleanroom centralizes project focus on the software specifications as the single source document on which to base all software design and all subsequent validation of requirements implementation.

Software Development

Cleanroom identifies rigorous and formal design as a necessary element for generating software whose correctness can be verified. A design method **[2]** based on structured programming theory is recommended for Cleanroom use. This method defines a limited set of primitives for capturing design logic, defining software structure and organizing the software's data. The primitives are used in a systematic and stepwise refinement of the software requirements and in the construction of a software design whose correctness can be assessed and confirmed at each step.

Software Correctness Verification

In the Cleanroom method, correctness is defined as the equivalence between a requirement and the design which supposedly implements the requirement. Designs are verified

using the functional technique for correctness verification **[3]**, first by the designer when constructing a design and subsequently by independent inspectors when reviewing the design. Correctness proofs in the functional approach work off the design structure rather than the embedded application logic, which allow the same proofs to be used across all design levels. With some algebraic manipulation, the question of correctness for a total software product can be reduced to the summation of the correctness proofs for the component parts.

Independent Software Product Testing

Software products are tested for two reasons - first, to ensure that the software correctly implements its design (structural testing) and, second, to ensure that the software satisfies its specified requirements (functional testing). Structural testing is primarily the responsibility of the software developer, while functional testing is generally performed by an independent organization.

In the Cleanroom method, only functional testing is performed since the correctness verification techniques, woven into the formal design method, satisfy all goals defined for structural testing. Functional testing is still required in the Cleanroom method for validating the implementation of the original requirements and a statistical approach **[4]** been defined and proven effective. Functional testing is driven by probability distributions which are defined against the requirements and generally track requirements usage in the software's operating environment.

Software Reliability Measurement

Cleanroom defines software reliability in terms of software mean time to failure (MTTF) which is a more meaningful measure for the user, which gives a positive quality indicator (longer MTTF is better) and which can be estimated prior to software delivery. When tied to a statistical testing approach, MTTF predictions during software development can accurately reflect subsequent operational experience.

Statistical Process Control

Cleanroom allows continuous process improvement through the effective use of reliability measurements taken during incremental releases of the software. Typically, incremental releases of software are staggered across a development schedule, so that MTTF readings from early releases can have dramatic impact on any combination of the specification, development and test phases. To gauge where corrective action is required in the process, the variance between the recorded and the target MTTF's can help identify what and how much correction is needed.

CLEANROOM INTRODUCTION STRATEGY

Introducing a software development method into an existing development environment is not easy and, in the case of the Cleanroom method, is further complicated because it also encroaches on the software specifier's and software tester's areas of responsibility. A clearly stated set of objectives must be defined which identify where and how much of the Cleanroom method is to be used. The planning for a particular software development entails Cleanroom training, identifying a tailored version of Cleanroom to fit the particular development environment and organizing checkpoints for re-evaluating decisions on technology selections. The training ensures a consistent level of understanding to plan the integration of the Cleanroom ideas into an existing development environment and to implement a problem solution. The successful Cleanroom project integrates the ideas into its environment and does not try to revolutionize its development process. The successful Cleanroom project also gives itself ample opportunity to change its process, as it gains experience, rather than stick with ideas which are failing for any number of reasons within the particular project environment.

Training in the Cleanroom Method

Training in the Cleanroom method is critical so that the project team has the depth of technical knowledge to apply the component techniques with conviction and effectiveness. The training is also necessary for the team's assessment and decision on which components of the Cleanroom method to use, because of the problem characteristics or

development environment. This training is best conducted in two steps, with formal instruction on the technical ideas followed by hands-on experience in applying those ideas to the project's particular problem.

The Cleanroom components to be covered in this training should include formal methods for software requirements specification, structured programming practice, the functional correctness model, statistical test methods, software reliability measurement and statistical process control. For each component, in-depth training on the theory and practice should be given to ensure that the selected method is understood and can be applied by the whole project team. In this process, aspects of a particular technique might have to be modified to fit the particular environment or to conform to organizational or contractual constraints and standards. In general, the details, on which aspects of a given method should become practice (assuming no loss of the kernel idea), tends to be less significant than the early establishment and consistent application of a practice. This should eliminate the endless debate on personal preferences within the team and should ensure a more effective use of the method.

Some of the Cleanroom techniques might be viewed as beyond the scope of the project definition or the abilities of the project team. In that case, serious consideration should be given to deferring the introduction of those techniques until a later project or phase of the current development.

In this training, formal instruction should be augmented with the attempted use of a particular method in solving the problem at hand. Each project member should have the opportunity to use the method, to decide its effectiveness to his assignment within the project and to make his suggestions on project practice. For the requirements specifiers and software developers, the hands-on experience should cover the specification, design and verification of some part of the top level design for the problem solution. For the software testers, the hands-on experience should include the attempted definition of a top level structure for the statistical data base to be used for the project's test sample generation. The objective of the hands-on experience is to confirm that the particular techniques can be used for the application and by the project personnel. This experience is necessary for organizing a tailored version of the Cleanroom method to be used on a project.

Michael Dyer
IBM Federal Sector Division
Bethesda, Md. 20817

Managing the Cleanroom Environment

July 31, 1991

Selecting the Cleanroom Components for a Project

Cleanroom is not an all-or-nothing method for software development but rather a collection of integrated components, which are intended to be used as a unit but can also be effectively used, individually or in combination. When starting a new project, a decision should be made on where the project should enter the Cleanroom process. For example, if measuring and using software MTTF is a critical requirement, then implementing the complete Cleanroom method should be seriously considered. Statistical testing should be viewed on its own merits as a functional test candidate, which can and has been used without the other Cleanroom components. Verification based inspections can be introduced into most software development processes, as long as software design is based on structured programming. Current Cleanroom experience reflects positive results with different approaches to introducing the Cleanroom method into a development organization and then evolving into the acceptance and use of the total method.

Because of its breadth, the Cleanroom method lends itself to an incremental introduction into a software development environment, where, in any given instance, only the techniques appropriate to a particular problem and a particular project team are selected and used. Force fitting a technique into a development situation is usually detrimental both to the success of the project and to the acceptance of the Cleanroom method within the development environment.

Planning the Introduction of Cleanroom

Adequate planning for the introduction of the Cleanroom method is critical to ensure against the potential for a project disaster, caused by the unwise or unsuccessful adoption of a particular Cleanroom component. Project managers are encouraged to establish milestones within the project schedules at which the progress of the Cleanroom technology transfer can be statused and assessed.

The number of milestones and their placement within a schedule will vary from project to project but, as a general rule, should appear frequently in the early part of the project schedule. A general rule of thumb is to schedule the initial milestones for each decision in

the first two-three months of a project, since these decisions shape the development process. Subsequent milestones for the particulars on the various decisions and for the necessary support (training, tools, consulting) should be scheduled in the first six months.

A specific goal should be defined for each milestone with a quantification of the technology transfer to the particular project. Project management should judge the progress being made in transferring the technology and decide whether changes are needed (eg. more training on specific technical topics) or whether the technology transfer should be stopped. In this latter case, the plan for reverting back to previously used methods should have been worked out, so that the recovery can proceed as effortlessly as possible. The planned schedule should contain sufficient flexibility to ensure the time and the resources to implement the recovery.

Generally, technology transfer would address developing the requirements specification with a formal method, integrating the functional correctness model into the baseline formal design method, eliminating development testing from the software process and implementing verification based inspections. From a test and reliability perspective, the transfer would address software testing with statistically representative user inputs and the estimation of software MTTF on a continuous basis during development. For each item, appropriate milestones should be defined to identify what was to have occurred, how success would be measured, what forward plan was to be activated, what tolerances on successful completion were acceptable and what recovery plan would be implemented in the unsuccessful case.

Milestones for Formal Specification Methods

For formal requirements specification, an initial milestone might be the completion of a top level software product specification, prepared by the lead engineer(s). A subsequent milestone might address the elaboration of the next level(s) of specification for the components of the software architecture. The intent of these additional milestones would be to involve all project software specifiers in the use of the formal specification method, to ensure that the specifier team can use the formal method and that the software developers and testers can understand their workproducts.

Michael Dyer
IBM Federal Sector Division
Bethesda, Md. 20817

The milestones would provide project management with the opportunity to assess whether the formal specification method could be used for the particular problem and by the particular staff. If the defined workproducts were not completed or unintelligible to the developers, testers and customers, then the effectiveness of the technology transfer would be suspect and some change in requirements specification is needed. Before reverting back to natural language specifications, the adequacy of the initial training, the availability of expert consultation and support tools and the levels of actual accomplishment should be reviewed. Since the specification is key to the project start-up, problems with applying the formal methods for specifications must be resolved, early in the schedule, and can not be allowed to linger into development. Either corrective steps are taken to get formal specifications on the project or the project reverts to established (ie. natural language) specification practice.

Milestones for the Functional Correctness Model

For integrating the functional correctness model with the existing design practice, an initial milestone might be the completion of a verified top level software design, which would give the first level decomposition of the specifications for the software architecture. The description might be a few pages of design language description, prepared by the project's lead designer(s). A subsequent checkpoint might be the completion of verified designs for the next one or two levels of decomposition. The objective for this milestone would be to give all the software designers on the project and opportunity to apply the functional model in constructing a verified design.

The milestones would provide project management with the opportunity to assess whether the design and correctness ideas could be applied by the lead and other software designers, in developing a solution to the particular problem. If the designs can not be successfully completed and verified to everyone's satisfaction by the planned milestones, then the effectiveness of the initial training in the functional correctness model, the completeness of the requirements specification and the commitment of the staff should be re-evaluated before proceeding. Any early problems with applying the correctness ideas need to be resolved with corrective steps (eg. additional consulting support, the use of analyzers to

guide verification, etc.). The alternative would be to stay with the established design practice, which probably means planning for more formal inspection and development testing at the completion of design.

Milestones for Eliminating Developer Testing

For eliminating development testing, an initial milestone might be the completion of the definition and planning of the library and configuration management procedures to support the delivery of code prior to its execution. A preliminary plan would be acceptable documentation for this milestone which would be prepared jointly by the lead software developer(s) and tester(s). A subsequent milestone might be the definition of inspection plans and milestones to ensure quality code delivery and of development procedures and tools to ensure that the design and code can created in a non-execution environment.

The milestones would provide project management with the opportunity to assess whether the project is serious about developing software without development testing and has put in place the tools and disciplines to facilitate this development approach. If satisfactory definition and planning is not completed by these milestones, then the commitment of the project to this objective should be reviewed. Testing by developers is a tradition which will not go away by decree but needs effective planning for it to happen (eg. separating the design and development from the target computer, limiting target computer access to testers, allocating a percentage (25-35%) of developer time to inspections, defining handover tests for acceptance of software into test, etc.). Unless this early planning and set-up is accomplished, the development will start on the wrong foot and the project commitment to this objective will probably evaporate. Either the appropriate development environment is organized to support development without developer testing, or the project should revert to its established development practice, making the necessary adjustments to accommodate developer testing.

Milestones for Verification Based Inspections

For introducing verification based inspections, an initial milestone might be the definition and planning of the inspection schedules, analysis tool and inspection format. A preliminary plan would be acceptable for this milestone, which was prepared by the lead software

developer(s). Subsequent milestones might be the completion of the requirements specification for the analysis tool, of the verified top level design for this tool and of the preliminary plan for testing the tool. This would be another opportunity to involve a cross-section of the project in applying the selected Cleanroom methods.

The milestones would provide project management with the opportunity to assess whether adequate preparation is being made for introducing the verification based inspection into the development process (eg. ensuring the allocation of sufficient personnel time, having the analysis tool available when needed, working out the formats of the inspection meetings, etc.). If there is project difficulty in completing these milestones, then the interest and commitment to introducing this new method should be re-examined and resolved (eg. subcontracting the analysis tool development). Without the early definition and planning, there will not be a smooth or problem free introduction of the verification based inspection. Either the necessary time is taken early in the project or the project should stay with its established formal inspection practice.

Milestones for Statistical Testing

For introducing statistical test methods, an initial milestone might be the definition of database organization, for generating the test samples. A preliminary description would be acceptable that defines a strategy for grouping the software inputs (eg. time, syntax, safety, etc.) and for organizing a selection hierarchy (eg. time periods, severity levels, etc.). The description would be prepared by the lead test engineer(s). Subsequent milestones might be the definition of the top few levels of probability distributions, the selection (or definition) of the generator support software and the encoding of an initial set of database entries. These latter milestones would involve a larger segment of the software testers and ensure acceptance of the statistical approach by the software testers.

The milestones would provide project management with the opportunity to assess whether a statistical approach to test sampling can be defined by the test organization and whether the mechanics of sample generation have been worked out. If there is project difficulty in meeting these milestones, then the applicability of statistical test to the particular problem needs to be reexamined and modified forms of statistical testing introduced (eg. multiple

user environments defined, existing traffic samples used in lieu of database definition, etc.). Either the effort is spent on defining a statistical approach or the project reverts to its established practice for requirements validation.

Milestones for Software MTTF Prediction

For integrating software MTTF prediction, an initial milestone might be the selection of appropriate statistical models and the definition of a prediction procedure. A preliminary plan prepared by the lead software tester(s) would be acceptable but would have to be integrated with a statistical testing plan. Subsequent milestones might include the installation and checkout of models, the definition of model validation procedures and the definition of MTTF prediction and assessment reports.

The milestones would provide project management with the opportunity to assess whether the project was set-up for MTTF calculations (ie. test interface, tools and procedures) and had defined a project role for software MTTF (eg. basic quality measure, control in a feedback process, etc.). If there is difficulty in completing the milestones, there should be a re-evaluation of the project's ability to do statistical prediction (ie. statistics background of staff, availability of models, etc.), of bottlenecks from the testing side (ie. statistical test plans, timing units, interfacing, etc.) and of the project's interest and commitment to doing something with the MTTF data. The fallback position would be use more traditional quality measures and not bother with statistical modeling.

CLEANROOM PROJECT MANAGEMENT

Project management with the Cleanroom method is not measurably different for project management when more conventional methods are used. One difference would be the tracking of the technology transfer milestones which provide project management with the opportunity to assess the introduction of the Cleanroom component techniques, to judge their acceptance by project staff and to measure their contribution to project productivity and quality goals.

Michael Dyer
IBM Federal Sector Division
Bethesda, Md. 20817

A second difference is the public visibility given to software quality by the early placement of software under formal configuration control and the continuous estimation of the software MTTF during development. Typically, software goes through various levels of review and inspection and various steps of developer testing, before it goes under configuration management. The theory is that enough effort (people and methods) has been given to removing errors, that the software is reasonably stable (small percent of remaining errors) and that the software can be given public (outside the project and, possibly outside the company) scrutiny without embarrassment. In the Cleanroom process, software is placed under configuration management prior to its first execution, which requires higher confidence and commitment from management in the Cleanroom's zero defect design strategy.

A third difference is the leadership and conviction that must be shown by project management in challenging accepted development practices and/or myths (eg. unit testing by developers, the ineffectiveness of randomized testing, the absurdity of software MTTF, the advanced mathematical background required for software verification and the futility of formal methods with changing requirements). Cleanroom offers counter intuitive ideas and methods which can and have been demonstrated to be practical and usable within the typical software development environment. Project management must ensure that staff skepticism in adopting these methods is overcome by providing the training, tools and consultation support to facilitate their effective use.

A fourth difference is to manage process improvement into the development effort. This requires observation and measurement of the process through the MTTF statistic, recognizing problems flagged by a constant or decreasing MTTF statistic and ensuring process correction via an increasing MTTF statistic. The incremental development strategy affords the measurement opportunities from which process corrections (eg. increased specification formality, broader participation in verification based inspections, etc.) can be defined for subsequent increment development and tracked for improvement effectiveness. The Cleanroom method provides a unique capability to project management for placing their software development under statistical quality control.

CLEANROOM IMPACT ON THE SOFTWARE PROCESS

Michael Dyer
IBM Federal Sector Division
Bethesda, Md. 20817

Managing the Cleanroom Environment

July 31, 1991

The introduction of the Cleanroom method would impact most steps in the software development life cycle, as shown in figure 1 which summarizes the role of the life cycle steps and the changes resulting from Cleanroom.

SUMMARY OF CLEANROOM IMPACTS
ON A DEVELOPMENT LIFE CYCLE MODEL

REQUIREMENTS SPECIFICATION
Function and Performance
but with
Usage and Build Statistics

SOFTWARE DESIGN/IMPLEMENTATION
Incremental Software Development
but with
Correctness Verification not Unit Test

INDEPENDENT SOFTWARE TEST
Integration & Test of Released Increments
but with
Representative Statistical Usage Samples

SOFTWARE ACCEPTANCE
Demonstrated Function and Performance
but with
Certified Software MTTF

Figure 1

Impacts on Software Specification

Software specifications define functional requirements and describe performance budgets that constrain execution time, size, etc. and environmental constraints such as interfaces, modularity, documentation, packaging and standards considerations.

With Cleanroom, the software specification is written with more formal notation to support correctness verification. Several acceptable methods are available such as box structuring techniques, formal specification languages (Z, VDM, etc.), and problem specific grammars. These formal methods force a closer analysis of the requirements and tend to minimize ambiguity, inconsistency and incompleteness in the resultant software specification.

Michael Dyer
IBM Federal Sector Division
Bethesda, Md. 20817

In addition to more formal specifications, Cleanroom forces the consideration of data on software usage and software construction to drive statistical testing. This includes the identification of software inputs and their expected usage probabilities to structure the test data bases. Any incremental release strategy must be elaborated to factor the planned availability of the software function into the test planning.

Impacts on Software Design

The major design impact is the introduction of functional correctness verification into the design process. The Cleanroom design ethic is one of requirements specification, followed by design of a solution to the specification, followed by verification of the equivalence between the design and requirements. Verification is integral to the design construction and imposes a control on the designer which gates the refinement of the software specification.

A second impact in the design step is the introduction of verification based inspections to provide an independent confirmation of the design correctness. The verification based inspection builds on the formal inspection practice **[5]** but re-orients the inspection to correctness confirmation rather than error detection. The reorientation is achieved through the use of design language analyzers which can determine the structure of the design and formulate the sequence and content of the questions to be addressed in inspections.

Impacts on Software Implementation

The impact to software implementation from the Cleanroom method will depend on the approach to software design. If design and verification are performed to full detail in the design step, then implementation becomes a transliteration of design notation into programming language notation.

An equally acceptable approach is to split the design refinement between the use of design notation and the use of the implementation programming language. The implementation

impact is that the software coding would now be performed stepwise, with each step verified for correctness and with verification based code inspections performed to confirm correctness.

Impacts on Software Developer Testing

The impact to this step is that it is no longer performed in the Cleanroom method. Close adherence to functional correctness verification ensures that all of the error detection situations addressed by developer testing are addressed in verification. With the Cleanroom method, the only reasons for software engineers to execute their software would be to check the feasibility or performance of newly defined algorithms, to exercise support software facilities and to confirm operating system services.

Impacts on Independent Testing

The Cleanroom method does not preclude testing because of software correctness verification, but rather relies on independent testing to validate that the software requirements were correctly implemented. Cleanroom impacts traditional testing by introducing statistical techniques. This impact on the tester has proven to be one of the harder obstacles to overcome in obtaining acceptance of the Cleanroom method. At the same time, statistical test techniques have the greatest potential for significant savings in the single most expensive part of software development.

CLEANROOM IMPACT ON THE SOFTWARE PRODUCT

Work on the Cleanroom method was originally started to improve the quality of delivered software and initial experience indicates that this purpose has been met. The quality improvement can be observed in quantitative terms from measures of software defects and in qualitative terms from improved software specifications, simpler software designs, faster error isolation and repair and fewer reported post-delivery problems.

Impact on Software Defect Rates

To get some feel for the levels of quality improvement being realized with the use of Cleanroom, two snapshots of reported data are provided. The second version of the COBOL Structuring Facility **[6]** was developed in five software increments. Error rates were measured from start (first software execution) through the completion of independent statistical testing and ranged from 1.4 to 5.7 errors per thousand lines of source code (ksloc), with an average of 3.4 errors/ksloc.

A similar picture of quality improvement was seen in the application of the Cleanroom method in the Software Engineering Laboratory (SEL) at NASA Goddard **[7]**. Error rates were measured from the start (again first software execution) through the completion of independent statistical testing and averaged 3.3 errors per ksloc. This compared very favorably to the 6 errors per ksloc which was the average experience of similar software developments in the SEL environment. In both the COBOL and SEL cases, the reported post-delivery errors were extremely small and measured in fractions of an error per ksloc.

Impact on Software Design Simplicity

One result experienced in all uses of the Cleanroom method was a demonstrated simplicity in the designs which were produced. Designers tended to be conservative in their designs. The result was a software design which satisfied the requirements (no less but no more) and used only known and easy to verify design ideas (nothing complicated nor exotic). This was seen repeatedly in the verification based inspections where 90% or more of a design could be confirmed in a straightforward manner and where design pieces, whose correctness could not be proved simply, were generally returned for further simplification.

The same simplicity was evident during the independent testing of the software where it could have been expected that the developer would need to execute the software to recreate error conditions and diagnose the source of failures. This turned out not to be the case **[6,7]** and developers were able to diagnose problems directly from their listings of software statements. In projects **[7]**, where the development organization had historical data on the time spent in finding and fixing errors, the reduction in effort was like an order

of magnitude, with repair cycles going from months and weeks to hours and days. This reduction is particularly remarkable since the software was always under formal configuration management, which imposed procedures and regulations on the fix and repair cycle.

Impact on Software Development Productivity

Software quality was the underlying objective of the work in developing the Cleanroom method. The added care in developing correct designs and the verification emphasis on inspections were new and different kinds of work, which were originally thought to add to the software design. Similarly, the added analysis in defining probability distributions and building statistical data bases for test sampling was originally thought to add some delta to the test effort.

A surprising result of the Cleanroom work is that software productivity did not go down and, in fact, increased in several cases. From the development side, design simplicity and the complete elimination of developer tester resulted in reduced effort that more than compensated for the work to integrate correctness into the software designs. In the case of the COBOL S/F and NASA SEL projects **[6,7]** the reported productivities were in the range of 750 lines of source code per labor month, which is three to four times higher than the average productivities, reported in the software literature.

REFERENCES

(1) M.Dyer
The Cleanroom Approach to Quality Software Development
John Wiley & Sons, Inc January 1992

(2) R.C.Linger, H.D.Mills and B.I.Witt
Structured Programming: Theory and Practice
Addison-Wesley 1979

(3) H.D.Mills
The New Math of Computer Programming
Comm ACM Vol.18 No.1 1975

(4) M.Dyer
Statistical Testing : Theory and Practice
Tutorial 7th International Software Testing Conference 1990

(5) M.E.Fagan
Design and Code Inspections to Reduce Errors in Program Development
IBM Systems Journal Vol.15 No.3 1976

(6) R.C.Linger and H.D.Mills
Case Study in Cleanroom Software Development
COMPSAC '88 Proceedings 1988

(7) S.Green, et al
The Cleanroom Case Study in the SEL
NASA Goddard SEL Series SEL-90-002 1990

Michael Dyer
IBM Federal Sector Division
Bethesda, Md. 20817

MANAGING IN THE CLEANROOM ENVIRONMENT

Prepared for
9th Pacific Northwest Software Quality Conference
Portland, Oregon
October 7,1991

Michael Dyer
IBM Federal Sector Division
Bethesd. Md. 20817

DISCUSSION OUTLINE

Overview of the Cleanroom Method

Strategy for Introducing Cleanroom

Cleanroom Project Management

Impacts on Software Process and Products

Lessons Learned

WHAT IS THE CLEANROOM METHOD

Technical and Organizational Approach to
Developing Software with Certified Reliability

Objectives

Focus on User Driven Definition for Reliability

Release Software with Known Reliability

Put Software Developed under Statistical Control

COMPONENTS OF THE CLEANROOM METHOD

Software Specifications

- Formal Notation for Function and Performance
- Usage Distributions and Construction Plans

Software Development

- Rigorous and Formal Design Method

Software Correctness Verification

- Correctness Woven into Design Process
- Verification Based Inspection Process

COMPONENTS OF THE CLEANROOM METHOD

Independent Software Product Testing

- Statistically Based Testing
- Test Samples of Representative User Inputs

Software Reliability Measurement

- Defined as Software Mean-Time-To-Failure (MTTF)

Statistical Process Control

- Continuous Process Improvement
- Driven by Software MTTF Projection

CLEANROOM IMPACTS ON A DEVELOPMENT LIFE CYCLE

REQUIREMENTS SPECIFICATION
Function and Performance
but with
Usage and Build Statistics

SOFTWARE DESIGN/IMPLEMENTATION
Incremental Software Development
but with
Correctness Verification not Unit Test

INDEPENDENT SOFTWARE TEST
Integration & Test of Released Increments
but with
Representative Statistical Usage Samples

SOFTWARE ACCEPTANCE
Demonstrated Function and Performance
but with
Certified Software MTTF

PROFILE OF EXPERIENCE WITH CLEANROOM METHOD

(Percent Of Cleanroom Projects Using Component Technique)

	Formal Specification	Baseline Design	Correctness Verification	No Unit Test	Statistical Testing	MTTF Prediction	Average Total Usage
Completed IBM Projects	33	100	66	100	66	50	69
Completed External Projects	0	100	0	100	100	0	50
Current IBM Projects	80	100	100	100	40	40	76
Current External Projects	100	100	50	100	50	0	66

STRATEGY FOR INTRODUCING CLEANROOM

Training in the Cleanroom Method

- Formal Specifications and Correctness Verification
- Statistical Testing and Reliability Modeling

Tailoring Cleanroom to Development Environment

- Expanding rather than Replacing Existing Process
- Considering Needs of Project and Staff

Planning the Inroduction of Cleanroom

- Checkpoints for Assessing Technology Transfer
- Introduction of Support Tools

CLEANROOM WORKSHOPS

Mixture of Theory and Practice

Selection of Three Forty Hour Courses

- Formal Specifications - Box Structure Method
- Formal Design with Rigorous Verification
- Software Certification - Reliability and Test Methods

Prerequisites

- Attendance by Project Teams
- Set Theory, Logic and Statistics Background

SOFTWARE SPECIFICATION WORKSHOP TYPICAL CURRICULUM

Problem Analysis

- Function Decomposition
- Function Allocation
- Requirements Traceability

Box Structure Analysis

- Design Principles
- Black, Clear and State Boxes

Specification Preparation

- Inspections and Reviews
- Incremental Development Plans
- Usage Distributions

ASSESSING TECHNOLOGY TRANSFER

Definition of Project Milestones

- Minimum of Two Per Technology
- Scheduled in First 3-6 Months of Project

Purpose of Milestones

- Quantified Assessment of Technology Acceptance
- Process Changes to Improve Technology Transfer
- Technology Work-Arounds to Ensure Project Completion

Candidate Assessments

- Formal Specification Methods
- Functional Correctness Model for Software Verification
- Elimination of Developer Testing Steps
- Verification Based Inspections
- Statistical Based Testing
- Software MTTF Prediction

SUGGESTED MILESTONES FOR FUNCTIONAL CORRECTNESS TRANSFER

Initial Milestone

- Completion of Verified Top Level Design Which Covers First Level of Requirements Decomposition
- Prepared by Project's Lead Designer(s)

Subsequent Milestones

- Completion of Verified Designs for Next One to Two Levels of Requirements Decomposition
- Prepared by All Project Designers

CLEANROOM PROJECT MANAGEMENT

No Change In Schedule and Resource Management

Cleanroom Unique Considerations

- Active Assessment of Technology Transfer
- Public Visibility with Early Software CM
- Leadership in Overcoming Skepticism on Technical Ideas
  (Correctness, No Debugging, Statistical Test, MTTF)
- Commitment to Statistical Process Control

CLEANROOM PROCESS IMPACTS

Improved Specifications with Formal Methods

Correctness Model Integrated into Design Practice

- Simplified Implementation from Design Attention

Developer Testing Replaced by Verification

Testing with Representative Usage Samples

Software MTTF for Tracking Product Quality

CLEANROOM PRODUCT IMPACTS

Product Quality Improvement

- More Prevention with Correctness Model
  Simpler Designs with Fewer and More Easily Found Errors
- Earlier Detection - 90+ % Errors Removed Prior to Test
- Order of Magnitude Reduction in Errors Found in Test and Field
  (3/ksloc during Test and < 1/ksloc post delivery)

Development Productivity Improvement

- Added Design Care Offset by Reduced Testing
  (2:1 Productivity Improvement Realized)
- Near Zero Life Cycle Maintenance

LESSONS LEARNED

About the Cleanroom Method

- Practical across Range of Applications
- Brings Formality to Software Development
  Mathematics and Functional Correctness to Design
  Statistics and Software MTTF to Test
- Puts Quality Focus on Customer Interests

About the Application of Cleanroom

- Tailorable to Existing Development Environments
- Usable by Software Practioners with Training
  Hesitant Acceptance by Developers
  Reluctant Acceptance by Testers
- Provides Both Quality and Productivty Improvement