SIXTH ANNUAL PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

September 19-20, 1988

Portland Marriott
Portland, Oregon

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

Chairperson's Message

Sue Strater

Welcome to the Sixth Annual Pacific Northwest Software Quality Conference. Software quality is important to customers today in differentiating software products to purchase. The conference is intended to make you aware of techniques, processes, tools, and methods that can give you an edge in producing high-quality competitive software products.

This year's keynote speaker, Edward Yourdon, will discuss the role of *"Software Quality Assurance in the 1990s."* His talk will show how the software quality assurance organizations can become a much more pro-active part of the development process.

This year's conference continues with parallel tracks on Project Management and Engineering topics. We have added one track as a forum for commercial vendors to give presentations on their products. The vendor exhibition is the other arena for attendees to see commercial products that are relevant to increasing software quality. The engineering sessions cover tools, testing, debugging, specification, design, and documentation - all important components of a product's life cycle. We have invited John Musa of AT&T Bell Laboratories to speak on Reliability.

Four workshops are being held on the second day of the conference. The topics are Project Management, Object Oriented Programming, Software Reliability, and The Economics of Software Quality.

The Conference Steering Committee hopes you find the 1988 Conference an enjoyable learning experience. We look forward to hearing any new ideas that you may have for future conferences.

Conference Committee

Front row: Kristina Berg, Craig Thomas, Sue Strater, Sue Bartlett. Back row: Bill Edmark, Len Shapiro, Mary Lawrence, Mark Johnson, Dick Hamlet, Steve Shellans.

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Sue Strater - President and Chair
Mentor Graphics Corporation

Dick Hamlet - Technical Program
Oregon Graduate Center

Craig Thomas - Vice President; Workshops
Mentor Graphics Corporation

Len Shapiro - Secretary; Keynote
Portland State University

Steve Shellans - Treasurer
Tektronix, Inc.

Bill Edmark - Exhibits
Intel Corporation

Paul Blattner - Board Member
Quality Software Engineering

Ben Manny - Board Member
Intel Corporation

PROGRAM COMMITTEE

Dick Hamlet, Chairman
Oregon Graduate Center

Sue Bartlett
Mentor Graphics Corporation

Greg Boone
CASE Research Corporation

Alan Crouch
Tektronix, Inc.

Sara Edmiston
Intel Corporation

Elicia Harrell
Intel Corporation

Warren Harrison
Portland State University

Mark Johnson
Mentor Graphics Corporation

PROGRAM COMMITTEE (continued)

Bill Junk
University of Idaho

Steve Shellans
Tektronix, Inc.

Steve Shimeall
Spacelabs Inc.

Sue Strater
Mentor Graphics Corporation

Bill Sundermeier
MicroCASE, Inc.

Lee Thomas
Mentor Graphics Corporation

Nancy Winston
Intel Corporation

WORKSHOPS COMMITTEE

Craig Thomas, Chairman
Mentor Graphics Corporation

Kristina Berg
Mentor Graphics Corporation

Paul Blattner
Quality Software Engineering

Ben Manny
Intel Corporation

Michael Meyer
Oregon Software, Inc.

EXHIBITS COMMITTEE

Bill Edmark, Chairman
Intel Corporation

Kent Mehrer
Mentor Graphics Corporation

Misty Pesek
QTC

Steve Shellans
Tektronix, Inc.

PUBLICITY

Dennis Schnabel
Mentor Graphics Corporation

PROFESSIONAL SUPPORT

Lawrence & Craig, Inc.
Conference Management

Liz Kingslein
Graphic Design

Decorators West, Inc.
Decorator for Exhibits

Authors

Hiralal Agrawal
Software Engineering Research Center
Department of Computer Science
Purdue University
West Lafayette, IN 47907-2004

E. G. Amoroso
AT&T Bell Laboratories
Whippany Rd., 14A-464
Whippany NJ 07981

Albert Baker
Department of Computer Science
Iowa State University
Ames IA 50011

Michel Bidoit
Laboratories de Marcoussis
Centre de Recherche de la CGE
Route de Nozay
91460 Marcoussis FRANCE

Gordon Blair
Rockwell International Corporation
Mail Code GA21
3370 Miraloma Ave.
Anaheim CA 92803

Frank Cioch
Department of Computer Science
and Engineering
Oakland University
Dodge Hall of Engineering
Rochester MI 48063

Rick Combest
Mentor Graphics Corporation
8500 SW Creekside Pl.
Beaverton OR 97005-7191

Lionel Craddock
Department of Computer Science
and Engineering
Hwy 52 & 37th St. NW
ABS-IBM 434/015-3
Rochester MN 55901

Christophe Dony
Laboratories de Marcoussis
Centre de Recherche de la CGE
Route de Nozay
91460 Marcoussis FRANCE

Benjamin Ellsworth
Hewlett-Packard Company
Mail Stop 5UR9
1000 NE Circle Blvd.
Corvallis OR 97330

David Hernandez
Technische Universitaet
Muenchen, FRG

Laveen Kanal
Computer Science Department
University of Maryland
College Park MD 20742

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

Jerry Luengen
J. Luengen & Associates
8000-D NE 58th Ave.
Vancouver WA 98665

Byron Miller
Director, Product Planning
KnowledgeWare, Inc.
3340 Peachtree Rd. NE, #1000
Atlanta GA 30026

John E. Moore
Hercules Aerospace
M/S N1ED3
Box 98
Masna UT 84044

John D. Musa
AT&T Bell Laboratories
Room 6E 11B
Whippany Rd.
Whippany NJ 07981

Jeff Offutt
School of Information
and Computer Science
Georgia Institute of Technology
Atlanta GA 30332

Thomas J. Ostrand
Software Technology Department
Siemens Research Laboratories
105 College Rd. E.
Princeton NJ 08540

James Purtilo
Computer Science Department
University of Maryland
College Park MD 20742

Pierre Robillard
Ecole Polytechnique/CS
University of Montreal
PO Box 6079, Station A
Montreal, Quebec CANADA H3C 3A7

William H. Roetzheim
The Mitre Corporation
3891 American Ave.
La Mesa CA 92041

John T. Rose
Department of Computer Science
Iowa State University
Ames IA 50011

Robert Shelton
CASEWare, Inc.
13050 SW Forest Glen Ct.
Beaverton OR 97005

Richard Sidwell
Rockwell International Corporation
Mail Code GA21
3370 Miraloma Ave.
Anaheim CA 92803

Eugene Spafford
SERC, Department of Computer Science
Purdue University
West Lafayette IN 49707

Sue Strater
Mentor Graphics Corporation
8500 SW Creekside Pl.
Beaverton OR 97005-7191

Bill Sundermeier
MicroCASE, Inc.
PO Box 1309
Beaverton OR 97075

Andrew Sung
Computer Science Department
New Mexico Tech
Socorro NM 87801

Ron Swingen
Mentor Graphics Corporation
8500 SW Creekside Pl.
Beaverton OR 97005-7191

Ilkka Tervonen
Institute of I.P.S.
University of Oulu
Linnanmaa, SF-90570
Oulu, FINLAND

Mladen Vouk
Department of Computer Science
North Carolina State University
Raleigh NC 27695-8206

Steven Wartik
Department of Computer Science
University of Virginia
Thornton Hall
Charlottesville VA 22903

Jonathan Weiss
AT&T Bell Laboratories
1 Whippany Rd.
Whippany NJ 97981

Elaine Weyuker
Courant Institute of Mathematics
New York University
251 Mercer St.
New York NY 10012

Nancy Willard
Attorney at Law
296 E. 5th Ave., #302
Eugene OR 97401

Jeanette Wing
Department of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

Edward Yourdon
Yourdon, Inc.
161 W. 86th St.
New York NY 10024-3434

Exhibitors

CASE RESEARCH CORPORATION
Contact: Greg Boone
155 108th Ave. NE, Suite 210
Bellevue WA 98004
206 / 453-9900

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

KNOWLEDGEWARE, INC.
Contact: Diana Felde
3340 Peachtree Rd. NE, #1100
Atlanta GA 30026
404 / 231-8575

MICROCASE, INC.
Contact: Bill Sundermeier
PO Box 1309
Beaverton OR 97075
503 / 690-1318

OREGON SOFTWARE, INC.
Contact: JoAnn Bertram
6915 SW Macadam Ave., Suite 200
Portland OR 97219
503 / 245-2202

PANSOPHIC SYSTEMS
Contact: Cindy L. Rogers
709 Enterprise Dr.
Oak Brook IL 60521
312 / 571-0818

POWELL'S TECHNICAL BOOKS
Contact: Joshua Eddings, Manager
32 NW 11th Ave.
Portland OR 97209
503 / 228-3906

PROMOD, INC.
Contact: Jann Fitzgerald
23685 Birtcher Dr.
Lake Forest CA 92692
714 / 855-3046

SOFTLAB, INC.
Contact: Paul Sutton
188 The Embarcadero
Bayside Place, 7th Floor
San Francisco CA 94105
415 / 957-9175

STATE-GRAPH, INC.
Contact: John Resing
PO Box 5697
Bellevue WA 98006
206 / 827-3548

Edward Yourdon

EDWARD YOURDON is the publisher of ***American Programmer***, a newsletter that analyzes software trends and their impact on the careers of American programmers. He is also an independent management consultant and author, and is widely known as the developer of the "Yourdon method" of structured systems analysis and design.

Mr. Yourdon has worked in the computer industry for 25 years, including positions with Digital Equipment Corporation and General Electric. He has worked on over 25 different mainframe computers, has been involved in a number of pioneering computer projects, and is currently deeply immersed in the personal computing revolution as well as research in new developments in software engineering.

In 1974, Mr. Yourdon founded his own consulting firm, YOURDON Inc., to provide a forum for educational, publishing and consulting state-of-the-art technology in the computer field. Over the next 12 years, the company grew to a staff of over 150 people, with offices throughout North America and Europe. As Chairman and CEO of the company, he oversaw an operation that trained over 250,000 people in major companies and government agencies around the world. The publishing division, YOURDON Press, produced over 50 technical computer books on structured analysis, structured design, and other software engineering topics; many of these "classics" are used as standard university computer science textbooks. In 1986, Mr. Yourdon sold YOURDON Press to Prentice-Hall; he now serves as the Series Advisor for the YOURDON Press list of books at Prentice-Hall and works with a network of over 1,000 authors around the world. At the same time, he sold the company's training and consulting operations to DeVRY, Inc.; YOURDON inc. was subsequently acquired by Eastman Kodak.

Mr. Yourdon is the author of over 100 hundred technical articles. He has also written 15 computer books, including a novel on computer crime, and a book for the general public entitled *Nations At Risk: The Impact of the Computer Revolution.* His most recent book is *Modern Structured Analysis*, a college-level textbook incorporating the latest developments in systems analysis, information modeling, CASE tools, and real-time systems design. Several of his books have been translated into Japanese, Russian, Spanish, and other languages, and his articles have appeared in virtually all of the major computer journals. He is a regular speaker at major computer conferences around the world.

For those who need to know about such things, Edward Yourdon received a B.S. in Applied Mathematics from MIT many years ago, when dinosaurs roamed the earth. Looking back on those years, he now wonders whether he should have studied basket-weaving instead.

Mr. Yourdon was born on a small planet at the edge of one of the distant red-shifted galaxies. He now lives in the Center of the Universe, New York City, with his wife, three children and seven personal computers, all of whom are linked together through an Appletalk network.

Software Quality Assurance in the 1990s

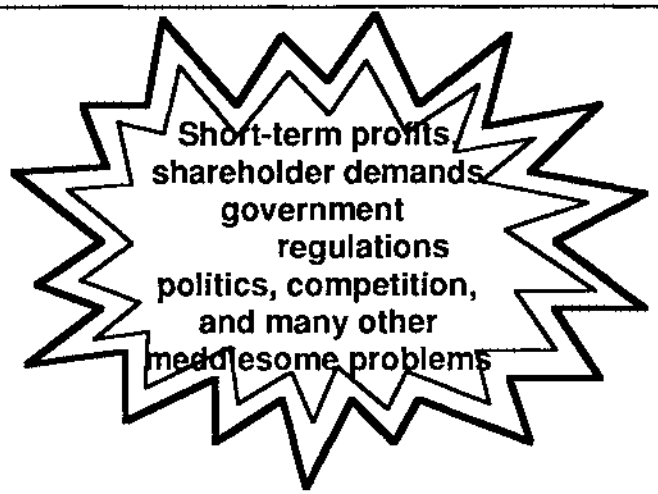
Edward Yourdon

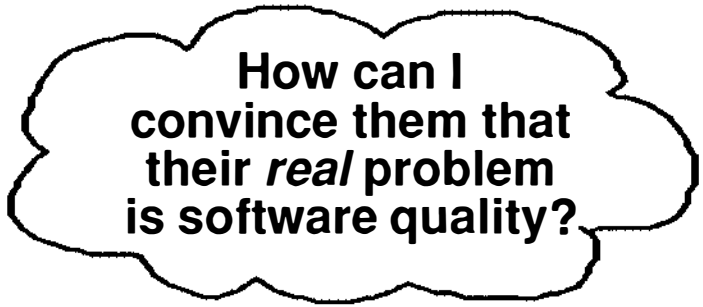
1988 Pacific Northwest Software Quality Conference
Portland, Oregon
September 19, 1988

The big problem as we enter the '90s

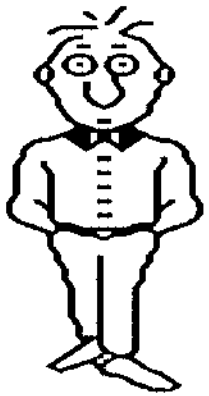
- Almost all software developers are aware of the urgent need for improving software quality...
- The real problem is that senior executives in most organizations don't understand what a serious problem it is...
- ...or what role they need to play to assure software quality.

The audience for software quality





Senior management



Software Quality:
An Executive Briefing

Your name
Date of presentation

Agenda

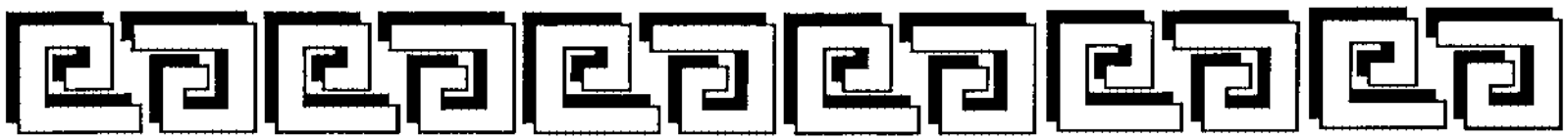
1. Isn't SQA just testing?
2. Why do we need SQA?
3. How well do we do it?
4. Improvements in the *technology* of software quality assurance
5. Improvements in the management process
6. Conclusions

What is SQA?

The classical view:

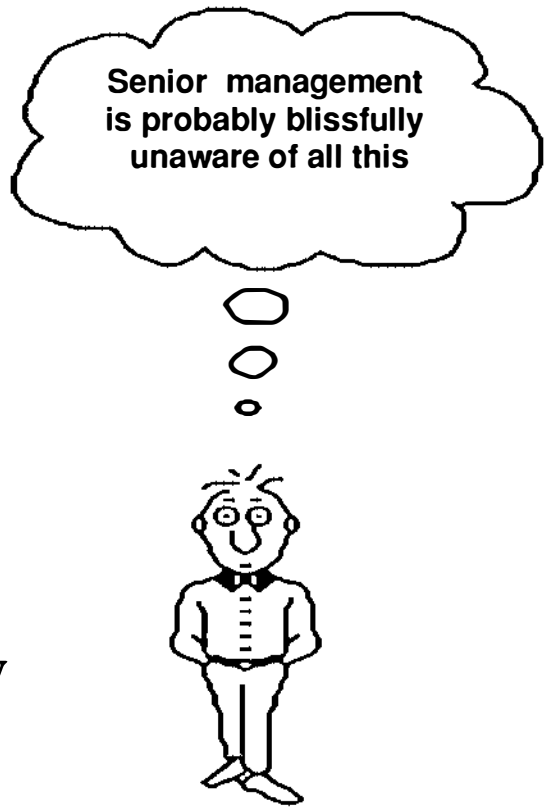
- ***SQA is just a fancy name for testing***

- ***Testing is what programmers do to find bugs in their programs.***



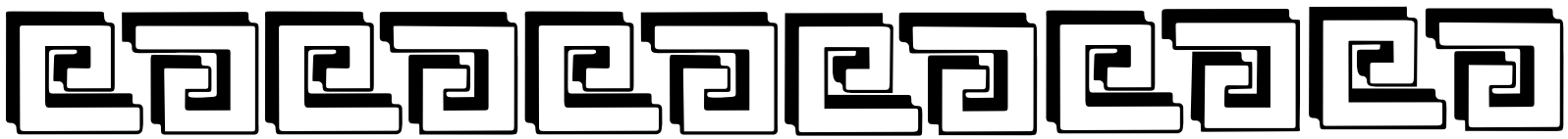
Problems with classical view

- Assumes people test their own programs
- Assumes that bugs are an independent life form
- Assumes that testing is done only once, at the end of the development phase
- Assumes that testing deals only with the *product*, and not with the *process* by which the product was built.



A better view of testing & SQA

Testing is one of the methods used by the Software Quality Assurance (SQA) group to ensure the quality of systems developed by the MIS organization



Some notes on this perspective

- Note that "quality" is more than just absence of bugs.
- Approx. 75% of U.S. organizations have independent SQA group.
- But recent survey by American Society of Quality Control indicates abysmal state of practice of SQA.
- Besides testing, other common SQA approaches are reviews/inspections and audits.

The abysmal state of SQA

- Most SQA personnel have no formal training in quality.
- Less than 25% of SQA people have professional SQA certifications.
- Most SQA groups do not use statistical approaches to quality control.
- Only 12% of U.S. companies use hard indicators to quantify quality.
- Only 10% use Pareto analysis.
- Only 6% use error-prone analysis.
- Very few companies use commercial SQA tools.
- Only 4% use reliability models (Lipow, Littlewood, Shooman, Musa).
- Only 7% use complexity models (McCabe, Halstead).
- Only 13% formally quantify the cost of their SQA programs.

The main point to make...

- There *are* software complexity models
- There *are* software reliability models
- There *are* statistical quality control models
- There *are* models showing the impact of "crunch mode" project management on quality and number of bugs in a system.

Your senior management should know how many errors remain in a system at all stages of development, and what it will cost them to put a "buggy" system into operation prematurely.

Why is SQA important?

- ***Because the consequences of an error are expensive and/or life-threatening.***

- ***Because low-quality systems threaten our organization's competitive position in the world marketplace.***

Why is it important? (cont'd)

- Senior management understands this intellectually
- But will they sacrifice quality for schedule and budget demands?

Software horror stories

- You may have some of your own...
- See "Risks to the Public," by Peter Neumann et. al., *ACM Software Engineering Notes*, Jan. 1988 and April 1988.
- Also Peter Neumann's "Some Computer-Related Disasters and other Egregious Horrors," *ACM Software Engineering Notes*, Jan. 1985

Some old horror stories...

- In 1979, the SAC/NORAD system recorded fifty false alerts, including a simulated attack whose outputs accidentally triggered a live scramble.
- An error in the F16 fighter simulation program cause the plane to flip upside down whenever it crossed the equator.
- An F18 missile thrust while it was clamped to the plane, causing the plane to lose 20,000 feet in altitude.
- The train doors on the computer-controlled San Francisco BART system sometimes open on long legs between stations.
- A NORAD alert from the Ballistic Missile Early Warning System detected the moon as an incoming missile.
- The Vancouver Stock Index lost 574 points over a 22-month period because of roundoff errors.

Software horror stories...

- First Boston faces a $10-50 million loss because of inaccuracies in its system that inventories mortgage-backed securities.
- First Interstate Bank suffered an "unexplained computer glitch" that caused a one-day delay in posting 3 million transactions, worth $2 billion, in December 1987.
- In February 1988, the Australian Commonwealth Bank posted all of its day's transactions twice, thus doubling all debits and credits.
- In early 1988, Bank of America acknowledged that it was abandoning a $20 million system after spending $60 million trying to make it work. 20 person-years of testing of 3.5 million lines of code had been invested.
- Rochester Telephone Corporation erroneously billed 4,800 customers for phone calls to Egypt when the computer system misread the number dialed.

More software horrors...

- A test program developed by a staff member at Wells Fargo inadvertently generated EquityLine statements in Feb 1988 with the following message:

  YOU OWE YOUR SOUL TO THE COMPANY STORE. WHY NOT OWE YOUR HOME TO WELLS FARGO? AN EQUITY ADVANTAGE ACCOUNT CAN HELP YOU SPEND WHAT WOULD HAVE BEEN YOUR CHILDREN'S INHERITANCE.

- A new AT&T computer billing system caused up to 2 million AT&T telephone customers across the country to be billed for payments they already made. Some accounts were mistakenly referred to collection agencies. (*Lafayette, IN Journal & Courier*, Jan. 29, 1988)

- A Westpac, Australia bank customer was erroneously charged with a $100 million overdraft, with interest accruing at $53,000 per day. The bank manager said, "Something happened with the computer and it went through..."

Another reason for SQA

- An organization's ability to function smoothly on a day-to-day basis depends on information systems.
- Its ability to make plans, decisions, and strategies depends on high-quality information systems.
- More and more products built by an organization contain software-intensive embedded systems.
- Services provided to customers by banks, etc. also depend on information systems.

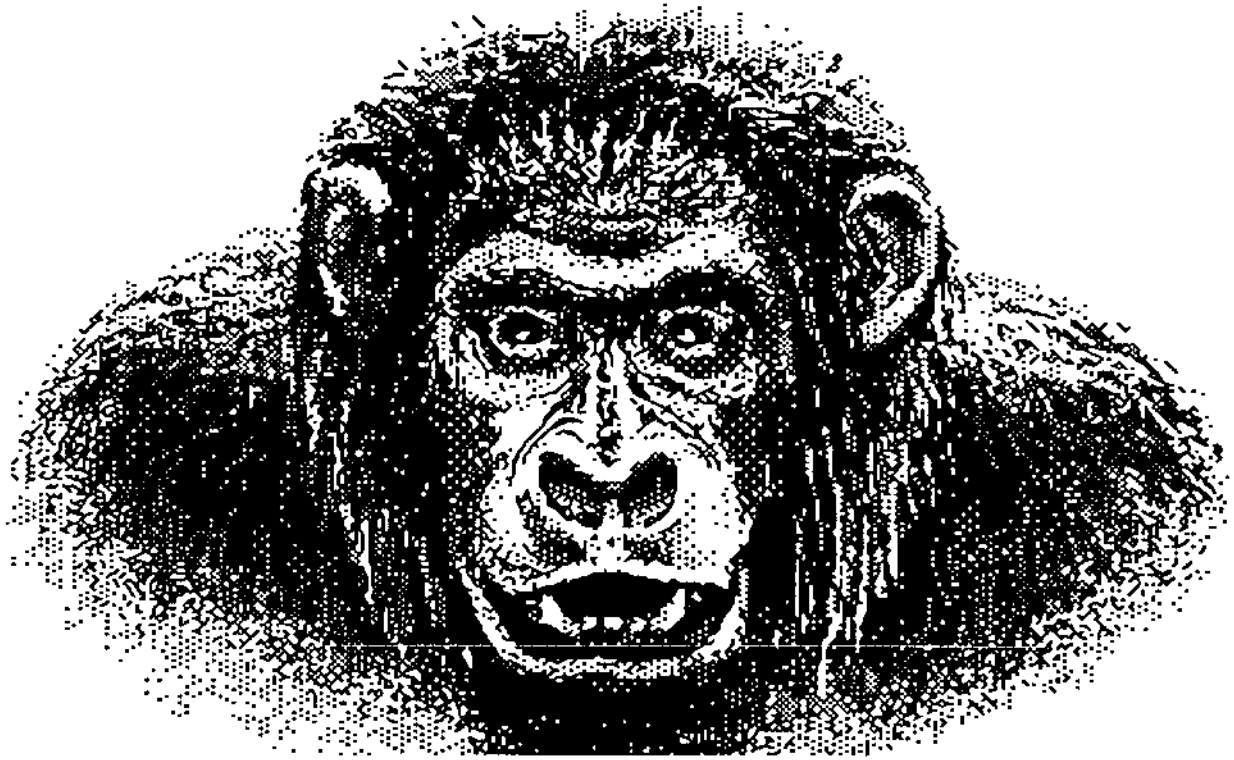
The bottom line...

- *An organization's ability to compete in the world market depends critically on the quality of its information systems.*
- *Other countries see our high-cost, low-quality systems as an Achilles heel that can be exploited.*
- *This does not just mean that the software "bodyshops" will face greater competition from cheap programmers in Singapore...*
- *It means that all organizations (and all employees) depend on high-quality systems for their jobs.*

Offshore Programming

- **Salaries 5-10 times lower than U.S.**
- **Chinese programmers earn $40/month**
- **Productivity equal or higher than U.S.**
- **Quality (measured in defects/KLOC) is 10-100 times higher.**
- ***Major* national R&D efforts underway in Singapore, India, Japan, Korea, etc.**

The American Programmer may no longer be king of the jungle



typical American programmer

How good is our quality?

- 50% of development budget is typically spent testing.
- Delivered systems have (average case) 3-5 errors per hundred lines of code. A "good" system is one with 3-5 errors per 1,000 lines of code. Best case seems to be 3-5 errors per 10,000 lines of code.
- Approx. 22% of maintenance budget is spent correcting errors. This is $72.89 per year for every man, woman and child in the United States.
- The chances of fixing a bug correctly on the first attempt are rarely better than 50%, and drop to 20% if significant amount of code is modified.

How can we improve SQA?

We must commit to senior management to improve the technological state of the art in SQA by:

- **Developing *better* computer-assisted proofs of correctness.**
- **Developing *better* theories of test coverage.**
- **Developing *better* statistical/probabilistic models of software reliability and software quality.**
- **Developing better *early* predictors of number of errors (e.g., by correlating with errors found in design phase).**
- **Developing better models of *human* causes of errors.**

How can senior mgmt help?

- Funding an active "technology exploration" activity
- Creating a formal SQA program if one doesn't exist.
- Investment in tools and techniques that will lead to better-tested, higher-quality systems.
- Most important: *a demonstrable commitment to quality.*

A Commitment to Quality

- Quality should be defined in specific, quantifiable terms that all of the players can understand.
- Organizational structure should be realigned, if necessary, so that those responsible for quality have the authority to make it happen.
- There should be a clearly understood reward system for quality—otherwise, why will anyone bother?
- Tradeoffs between quality and other constraints (budget, schedule) should be clearly articulated.

The End


That's the end of the presentation to senior management ... but there's more to talk about.

What if they don't listen?

- The quiet revolution
- The open revolution
- Voting with your feet.

The quiet revolution

- **Basic idea: don't change the management structure, change the culture.**
- **Philosophy: improve quality for personal satisfaction, even if the organization doesn't know about it, or doesn't approve it.**
- **There are a thousand minor day-to-day decisions that senior management never knows about.**

30

The open revolution...

- If they ask you to “speed up testing...”
- If they say, “Don't worry about a few bugs...”
- If they say, “We'll pin down the specs later, just start coding...”
- If they say, “Don't worry, this is just a beta test version...”
- If they say, “I don't care if there are bugs, we've got to have this system delivered by January 1st...”

Vote with your feet

- **Time is running out...**
- **If your organization can't be changed, leave it.**
- **Seek employment with an *exemplary* organization.**

If all else fails....

Project Management 1.

"Risk Management in Software Engineering"
Jerry Luengen, J. Luengen & Associates; Ron Swingen, Mentor Graphics

"Control Charting in Software Development"
Lionel L. Craddock, IBM Application Business Systems

"Five Components of a Software Quality Assurance Paradigm"
Patrick H. Loy, The Johns Hopkins University

CONTINGENCY MANAGEMENT IN SOFTWARE PROJECTS

Jerry Luengen
J. Luengen & Assoc.
8000-D N.E. 58 Ave.
Vancouver, WA 98665

Ronald K. Swingen
Operations Manager, Advanced Products Division
Mentor Graphics Corporation
Beaverton, Oregon

ABSTRACT

On January 28, 1988, the Vancouver Columbian newspaper carried a story about the Bank of America abandoning a trust accounting system after spending $60 million over several months in an unsuccessful attempt to fix the system. Most of us know of other failed projects and have come to believe that unsuccessful or at best, marginally successful, projects are considered common. Why then are there so many unsuccessful or marginally successful software projects?

The authors believe that an important dimension of project management is being neglected. Further, the techniques, tools, and educational materials for this neglected dimension of planning remains undeveloped. It's time we put some emphasis on contingency management, the neglected dimension of software project management. This paper extracts the latest contingency management ideas in the professional literature, explains the mechanics of constructing a program for minimizing risk in sofware projects, and provides insights from the authors' experiences in managing the risk areas of projects.

BIOGRAPHICAL SKETCHES

Jerry Luengen is a Project Management Consultant and principal of J. Luengen & Associates. Over 24 years of experience in a variety of industries and projects, he has held every position available in the field of project management. Currently, Mr. Luengen presents project management workshops, provides litigation support, and is an advisor to several Fortune 500 companies. He founded the Alaska Chapter of Project Management Institute and regularly presents project management workshops and papers at regional symposiums.

Ronald Swingen is the Operations Manager of the Advanced Products Division of Mentor Graphics Corporation. As Operations Manager he is responsible for managing software releases and the software development environment. His organization also provides Quality Assurance for the division's products. He has over 20 years of software development and project management experience and has worked for International Business Machines, Tektronix and Intel prior to joining Mentor Graphics. Mr. Swingen has also been an instructor and consultant on project management and organizational effectiveness.

CONTINGENCY MANAGEMENT IN SOFTWARE PROJECTS

"The best laid schemes o' mice and men gang aft a-gley."
Robert Burns

Contingency management is the combination of risk analysis and contingency planning, a major component of project management and one that is frequently ignored. Contingency management is an obligation of every Project Manager for it is the only way for the Project Manager to manage unplanned adverse events which may prevent a project from succeeding. There are techniques, procedures and a body of knowledge that may be employed to manage anticipated project problems. This paper will explain the basics of contingency management by describing both a systematic approach to reducing the impact of unplanned adverse events, and the mechanics of constructing a comprehensive program for minimizing risk in software projects.

Definitions of Risk Analysis and Contingency Management Terms

One possible reason why contingency management has been ignored may be lack of knowledge of the key terms and their definitions used in contingency management. The definitions below seem to be the most widely accepted within project management.
Certainty - absolute knowledge of all possible risk event outcomes.

Contingency - "Something whose occurrence depends on chance or uncertain conditions; a possible, unforeseen or accidental occurrence." (from Webster)

Contingency management - the project management function devoted to the identification, planning, control, and mitigation of negative events upon a project. Contingency management may also be thought of as those steps directed toward ensuring a successful project in spite of all that can go wrong.

There are some other words that are being used in place of contingency management, e.g. risk management, risk evaluation, risk assessment, etc. As you can see by the definitions, any term incorporating the word risk doesn't incorporate the contingency planning activity. The authors prefer contingency management as the term or label which best expresses the global management of anticipated, but specifically unknown, adverse events which may happen to a project.

Contingency planning - planning to prevent, avoid or reduce the impact of tomorrow's anticipated, but specifically unknown risks. Contingency planning is the process of identifying risk areas, assigning probability of occurrence, determining severity of impact, identifying causes and possible mitigating actions, and choosing the best response to the possible risk event.

Impact - The impact of some adverse event on your project may be measured first by its effect on the project goal or end product and, second, by it's effect on one or more of the project parameters: cost, time, quality, or scope. Ireland and Shirley, in their paper "Measuring Risk In The Project Environment", define five categories of "consequence character" (impact). Their scheme is included below for two reasons. One, it is a good example of a five category valuation scheme. Second, we believe that any responsible Project Manager would select any risk event having a rating in the three top categories i.e. very severe, severe, and moderate, for further review. In effect, their system is a three category valuation system: major impact, little, and very little. The category labels are irrelevant. Although the authors of this paper prefer high, medium, and low for

CONSEQUENCE CHARACTER	CRITERIA
Very Severe	• A failure of an activity, in itself, that will cause the project to fail to meet its goal.
Severe	• A failure, in combination with the failure of one or more activities, that will cause the project to fail to meet its goal.
Moderate	• A failure, in itself, that will cause a major impact on cost, schedule, and / or quality of the system / project.
Little	• A failure, in itself, that will cause only a minor impact on cost, schedule and / or quality.
Very Little	• A failure, in combination with the failure of one or more activities, that will cause only a minor impact on cost, schedule and / or quality.

"Table 3" - from "Measuring Risk in the Project Environment" by Ireland and Shirley

impact designators, it makes just as much sense to use extreme, moderate, and low; or any other three labels having the same connotation. Don't quibble over labels; choose impact labels and their definitions to fit your projects.

Probability - the likelihood of an event occurring. A 50% probability means that an event is as likely to occur as not. Probability is usually expressed as an exact two digit decimal between 0 and 1. The closer the probability is to 1.0, the more certain is its occurrence.

Risk - the potential for unwanted, negative consequences of an event. Risk does not occur under conditions of certainty. Risk concerns those activities or outcomes for which a probability value can be assigned or estimated.

Risk analysis - the systematic assessment of probability and consequent impact of possible adverse events.

Risk assessment - the product of two things: probability of occurrence and the consequences or impact of the occurrence on the project.

Uncertainty - refers to outcomes for which a probability value cannot be assigned.

Common Contingency Management Problem Areas

Contingency management has some problems peculiar to project management. Here are some observations and comments about the most common problems.

1. Most project risk evaluation does not concern risk, which by definition has known probabilities, but the evaluation of uncertainty. Much of what we generically designate as risk in a project will, on closer inspection, turn out to be uncertainty. We are uncertain as to the probability of the occurrence and uncertain as to the severity of the consequence.

2. There is a problem with the quantification of uncertainty. It would be very helpful if we could reduce a subjective evaluation of probability and impact to precise numbers that could be neatly manipulated in subsequent decision making schema. For example, suppose that some amount of risk can be attributed to the lack of a precise user specification. Can we reduce our subjective judgement of risk in this example to a two place decimal between 0 and 1 that indicates probability? Can we evaluate the impact on the project in terms of dollars? The answer is yes and no. We can estimate, but never know for sure, the exact figures for probability and impact until we actually experience the results of an imprecise user specification. We need to keep in mind that any translation of subjective value into an exact number carries an inherent variance that is an unknown plus or minus deviation from actual. Therefore, any attempt to use these quantified uncertainty numbers in further calculations of risk levels, cost exposure, and as a basis for actions should be viewed with skepticism and, certainly, reviewed with care. They are approximations or guides, but not necessarily an accurate depiction of the future.

3. The tools that exist are not being used. Applications of the basic tools and techniques of contingency management are described in subsequent sections of this paper.

4. Insufficient planning time is spent on analyzing project risk or uncertainty. We recomend that at least 20% of total planning time be devoted to examining what can go wrong.

5. There are many techniques such as decision trees that deal with risk, but few that deal with uncertainty. Two techniques that deal with uncertainty are described in the Recommended Approach to Contingency Management Section of this paper.

6. Risk and uncertainty evaluation techniques, excepting the techniques proposed herein, are not easy to use. Our opinion is that unless tools and techniques are easy to use, they don't get used.

7. The assignment or estimation of risk probabilities is difficult. By definition, a project produces a unique, one-time, product. How accurate is a probability estimate for a series of events and activities which produce unique project products?

8. In a similar vein, there are dissenting statistical views that ask how valid are probabilities on projects where there is no previous history? Are probabilities valid without a large number of similar occurrences?

9. Quantifying risk is the translation of a subjective opinion into an exact number. There are several methods of quantifying risk that have been discussed in the references. Most are tedious and revolve around the conversion of questionnaire answers into some sort of rating scale. It appears that the assigning of values becomes the central focus of risk assessment and the major part of contingency planning. In reality, when all is evaluated and the rating is done, the result is that some aspect of the project is judged to have a certain degree of risk or some exposure to possible adverse consequences.

Determining the degree of risk is only an intermediate step. Each of the following steps also introduce subjective evaluations or "best judgement" valuations which, cumulatively, reduce the value of exact quantifications. The only value of a risk assessment is to focus attention on the most likely "high impact" and "high probability" areas. These high risk and probability areas are then culled out for further review. Therefore, the authors have this advice:

a. Don't over emphasize the reduction of subjective information to exact numbers. You end up agonizing over the difference of 10% in two place decimals which is nonsensical after considering the inexact methods used to arrive at the numbers.

b. Three categories: high, medium, and low are adequate for degree of risk evaluations. Four or five categories are feasible, but we would not recommend anything over five.

10. Dealing with the uncertainty of time estimates is always raised in any discussion of project risk evaluation. Uncertainty is inherent in the definition of estimates and, in particular, time estimates. The authors feel that much of the uncertainty of about time estimates could be treated with better estimating practices which is beyond the scope of this paper.
The other part of the uncertainty associated with time estimates deals with their individual and cumulative effective on the project. Generally, each project activity duration estimate has a range of accuracy and that accuracy varies from activity to

activity. The best way to consider the total effect of these estimates having varying accuracies is to use a project management software program to model the project. By inserting varying duration values for activities causing concern, we can see how sensitive the project finish date and critical path are to changes in one or more activity durations. We can also test to determine whether secondary paths could become critical and determine which activities have a high potential to become bottlenecks on one or more paths in the network.

For the purposes of this paper, the authors recommend that you consult Clifford Gray's book "Essentials Of Project Management" or others for a more exhaustive treatment of how to handle this specialized area of project uncertainty. If the uncertainty of time estimates is relatively small and unimportant (as we think it should be with proper estimating and contingency planning), then consider doing a minimum of modeling (or "what iffing"). On the other hand, if the cost or importance of possibly being wrong on a time estimate is great, then investigate the use of PERT, using a three-time estimate PERT calculation, and doing PERT simulation.

These are the common problem areas in contingency management. An awareness of these problem areas should not diminish your enthusiasm to use contingency management in the pursuit of better project management. It only means that contingency management is still an inexact science. We just need to keep in mind the limitations of the techniques employed.

Recommended approach to Contingency Management

The number one obligation of a Project Manager is to ensure project success. "Make it happen" is the dictum. You can't possibly do that without spending some part of your planning time considering "what could go wrong" and defining positive steps to avoid, minimize, or protect your project from, sometimes, easily anticipated potential problems. Risk evaluation is a team effort so involve the project team and any other support staff who may lend wisdom to evaluating project risk.

Project risk evaluation should be done in at least **two phases**.

1. Evaluate the project on a global basis during the project organization stage of the project planning phase.

2. Evaluate each project task, activity, or work item well in advance of execution or actually doing the task.

Any evaluation of project uncertainty should include consideration of at least these areas.

- Technical or design unknowns
- Peripheral equipment
- Quality specifications
- Subcontractors
- Terminations
- Long lead items
- Market risks
- Trade show support
- Beta site support
- Budget and cost estimate
- Executive support or project priority
- Project Manager's charter to manage
- Contractual risks or penalties
- Critical personnel or skills
- Technical exposures
- Client/User requirements
- Training
- Host environment availability and reliability
- Political conditions -inter and intra- project
- Development environment / tools / language
- Undefined team/organization interfaces & responsibilities

Consider using an adaptation of Kepner and Tregoe's potential problem analysis as outlined in their book, The Rational Manager. Their technique uses a series of prompting questions and a procedure paraphrased and expanded as shown below.

Ask the project team, "What can go wrong with X?"; "What are the major problems associated with X?" and "What is a worst case situation for X?" where X is one of the categories mentioned above - host environment, budget, schedule, etc. Try and relate these to various project work items. Use a prompting style of posing questions. Ask the same question in different ways. The idea is to trigger some response on the part of the audience that is helpful input. If you don't feel you are getting enough response with the questions above, try these:

- Who is in control of this event (if anyone) and can it be managed?
- When during the project is this event of consequence?
- Is there a dependency of this event on some other event and if so, what is that dependency?

Then for each event ask: what is the impact on the project - low, medium, or high? What is the probability of occurrence - low, medium, or high?

Next, screen the answers and look for problems having high impact and a high probability of occurrence - particularly those affecting a task on the critical path.

Then ask for each of the high impact - high probability events (high-highs): what are the likely causes?

Next, what are the possible actions to resolve each possible risk? Actually, Kepner and Tregoe says there are three possible classes of actions: preventive, protective, and adaptive. Then choose the best possible action that prevents the problem from happening first. If you can't prevent it, what actions will protect you? If you can't prevent

it or protect yourself from the problem, how can you best adapt to it?

Choosing the best action necessitates consideration of additional factors. Generally, the best action is one the minimizes impact on:

- your project goals
- resources
- ethics
- applicable policies
- project budget
- project schedule
- conflicts of interest

Frequently, there is more than one appropriate action. Do whatever is necessary to resolve the problem as efficiently (least cost and impact to the project) as possible. Usually, choosing a best action has an impact on one of the key project parameters: cost, time, scope, or quality. Revising a major project parameter requires definite substantiation and explanation of additions or modifications. If no rationale is provided, then prudent management will not support alteration of a project baseline parameter.

Summary of the Technique

The above procedure works best when applied globally in the initial organization phase with the project team doing the analysis together. Note that an integrated approach has been taken with respect to risk assessment and initial contingency planning . We have also recommended that this be accomplished in two phases. Whenever possible the project leader should facilitate the risk evaluation session. Do not farm out risk evaluation to one individual. The greater perspective of the project team is required as well as other experienced and knowledgeable individuals within the company. The analysis should include:

1. Critical path tasks
3. Key or critical events (what could go wrong)
4. Problems that have been encountered in the past
5. Other tasks as time permits.

For each of the above, these are the suggested priority of investigation.

1. High impact and high probability of occurrence
2. High impact and medium probability of occurrence.
3. Medium impact and high probability of occurrence.
4. Medium impact and medium probability of occurrence.
5. Combinations of low impact or low probability.

Record the results of the analysis on a form similar to the Risk Evaluation Analysis Form described in Appendix A. A reasonable analysis of a fairly large project can be completed in less than two hours.

After all of the global risk assessment and contingency planning is done, there is one final step. Review whether the project is still feasible. There may be additions to the cost, time, quality, and scope parameters such that the project is no longer feasible. Or, it may still be feasible with approved additions to budget, schedule, scope, or quality. Whichever the case, a feasibility review should be done as a final check.

The Risk Tracking Aspect of Contingency Management

Another aspect of contingency management is risk tracking. We have proposed above that project risks be identified, assessed, and a contingency plan developed for those risks that have a high impact and a high probability. Sometimes there is no action until a triggering event occurs. The triggering event should identify what constitutes a need for action and when to expect such an event. The triggering event sets in motion a predetermined series of actions according to the contingency plan. The action associated with a triggering event may range from the initiation of an action plan to the removal of theneed to continue tracking the risk. Each risk should have associated with it a triggering event, and that triggering event should be tracked along with all other project milestones and required actions. Periodic reviews of the project risks should be conducted to assure that new risks are acknowledged and assessed, that risks that are no longer of concern are removed and that the impact and probability of previously identified risks are still valid. The length of time between these reviews should generally be one month, but could be done on a quarterly basis for projects running more than one year.

Reporting project progress must include an update of the risks being addressed by the project. Risk reporting would cover the status of any action plans currently underway as well as any risks that have moved into or out of the high impact and high probability category. The format of risk reporting can be either a narrative style of reporting or could be summarized on a form such as the Project Contingency Planning Summary shown in Appendix A. A final element of the risk reporting component of the project report is whether or not the project is still feasible.

Case Study

An example of a contingency plan is shown in Appendix B. The example describes a software release of a product that was extremely important to the company developing the product. Section 9.0 of the Release Project Plan was the section dealing with contingency management. The time span covered by the release was nine months which included a very important trade show, the Design Automated Conference (DAC), as well as Strategic Planning for the corporation and Field Application Engineer (FAE) Training. Each of these last two items were being done at the same location where the development work for the release was being done. The release also was scheduled just as the major hardware vendor was starting to ship hardware fixes for a transceiver problem. The only risks included in the initial plan were those judged to be of High Impact and either High or Medium Probability. Of these initial risks, DAC and theComputer Hardware Changes did indeed require implementation of their related

Action Plans. The remaining risks did not require action other than for tracking and removal once the dates for the events had transpired. The Release was completed on time, which would not have been true had the DAC and hardware impacts not been properly managed.

A Total Approach to Contingency Management Is Important

Contingency management, fortunately, is receiving more attention as part of project management. Professional Project Managers are beginning to see the merits of spending significant planning time in looking at what could go wrong with a project. In addition to contingency management within individual projects, consideration must be given to the project management infra-structure and it's impact on individual project management. There are at least three different levels in the corporate hierarchy where contingency management in individual projects may be enhanced. These levels are corporate, group/division and project. The concern at each level should be "how can adverse consequences be prevented"?

The following suggestions provide guidance for reviewing project risk or uncertainty at each level in the corporate hierarchy.

Corporate

1. Reduce corporate project risk with project policies and guidelines. Examples are:

    A. All projects shall be planned, managed, and controlled with due regard for contingency management, time, cost, quality, and performance objectives such that the project client and corporate objectives are met.

    B. Project Impact statements shall be issued by the Project Managers when project forecasts differ significantly (greater than 10%) from baseline objectives.

    C. All Groups/Divisions shall promulgate project execution manuals, appropriate to their size and mission, which spell out how projects will be processed, roles and responsibilities of all project staff, and standard techniques and tools that should be used in the practice of project management.

    D. Projects will be budgeted, planned, and implemented according to a priority and ranking system.

Group/Division

1. Develop and maintain a project execution manual which defines project contingency management procedures.

2. Define a series of project processing checkpoints or approval points such that contingency management is a part of the approval review.

3. Allot time for managerial review of contingency management.

4. Recognize that time and cost contingencies are a necessary part of project schedules and budgets.

5. Establish requirements for prototyping when prototyping is necessary.

6. Define the Project Manager's authority with respect to initiating action plans and latitude for decision making.

7. Assure that project change control mechanisms exist.

8. Assure that project baseline deviation impact statements be issued as required.

9. Emphasize code re-usability - modularity within software development projects.

Project

1. Conduct global project risk evaluations during the project organization phase.

2. Do risk evaluations for each critical path task.

3. Emphasize that all team members have an obligation for contingency management.

4. Document project cost estimate and scheduling assumptions.

5. Spend at least 20% of project planning time on risk assessment and action plans.

6. Keep contingency time and budget dollars visible.

7. Institute project change control procedures.

8. Ensure that the project client, sponsor, and bill payer (all could be one and the same) are part of the project team and that their objectives, desires, priorities, etc. are documented.

The Benefits of Contingency Management

The techniques described in this paper provide a methodology whereby Project Managers can be reasonably certain that most conceivable adverse consequences will be prevented or reduced in severity. The authors have proposed a systematic approach to anticipating potential crises and heading them off or reducing their impact. Individual Project Managers as well as managers in the corporate hierarchy have a common interest in promoting a total approach to contingency management. And, if you will do that, then these benefits will become apparent:

- There will be more successful projects.
- Less management by crisis.
- Improved quality of project end products.
- More peace of mind from knowing that you have done all that you can do to prevent and mitigate crises.

Strive to develop a systematic, structured approach to contingency management that is simple, easy to apply and easy to teach. If it isn't easy, it doesn't get used.

One characteristic that successful Project Managers demonstrate is their ability to stay ahead of the project in terms of dealing with crises. What this usually means is that the number of true crises (i.e. unplanned severe impact events not anticipated in our contingency planning) is held to a minimum. Additionally, senior management is usually very alert to the potential for disaster in projects and is extremely interested in the contingency management aspect of the projects they sponsor or approve. They are more inclined to provide additional resources to a project that has only one or two crises, rather than one that is always in a crisis mode.

If you will conscientiously set aside 20% of your planning time for contingency management, we are convinced your projects will go smoother. Our approach to contingency management will not entirely eliminate project problems, but, it will resolve most of them . . . in advance . . . on your terms.

Appendix A. Forms

1. Risk Evaluation Analysis

2. Project Contingency Planning Summary

RISK EVALUATION ANALYSIS

(Use this format for evaluating project risks)

Risk Area: (Description)

Probability of Occurrence:

Impact of Problem (if left untreated)
- Cost:
- Time:
- Quality:
- Scope:
- Other:
- When is corrective action req'd?

Most Likely Cause:

Recommended Action:
- Action:
- Why this action?
- Alternatives:

Consequences of Recommended Action?
- Cost:
- Schedule:
- Quality:
- Scope:

Disposition:

Follow-up:

PROJECT CONTINGENCY PLANNING SUMMARY

DATE _____

PROJECT _____ ENGINEER _____

CONTINGENCY CONCERN	Area or Activity Number	Date Action Proposed	PROPOSED ACTION (By Whom)	COORDINATE HOW? * Department. * Cost * Resource * Timing

Appendix B. Risk Management Example

Section 9.0 Risk Management

9.1 Design Automation Conference (DAC)

Impact: High
Probability: High
Triggering Event: Resources reassigned from the Release to DAC support, either directly or indirectly or DAC ending date (6/23/85) is passed.

Action Plan:
Project Manager will discuss any reassignments with the functional manager making the reassignment to determine the length of time for the reassignment and the impact of the reassignment. If the impact will cause the Release to slip, then the Project Manager and the functional manager will immediately discuss the situation with the Design and Analysis Division (DAD) General Manager (Project Sponsor) and the functional manager's manager. No slippage of the Release schedule will be allowed without the approval of the DAD GM.

9.2 Computer Hardware Changes

Impact: High
Probability: High
Triggering Event: Arrival of the new hardware boards. Status to be checked weekly to determine arrival date of the boards. If the boards are delayed past June 1, supplier needs to be contacted to determine status of the shipment.

Action Plan: The transceiver fixes must be installed as quickly as possible to ensure network stability. The priority of installation is 1)Network Servers, 2) Nodes of Development Engineers and Quality Assurance Engineers working directly on this Release and 3) Other nodes. Priorities of installation should NOT be to customers, since customers are not running networks as large as ours and are therefore not encountering the problems that this hardware change will correct. Close communications are mandatory between the Shipping and Receiving Manager, Systems Administration Manager and the Project Manager to ensure that installation begins as soon as the boards arrive. If shipment is delayed past June 1, Project Manager should contact supplier to determine what the cause of the delay is and to pursue a course of action that will result in the boards arriving as quickly as they possible can.

9.3 Corporate and Division Strategic Planning

Impact: High
Probability: Medium
Triggering Event: Initiation of Strategic Planning Process and completion of the process.

Action Plan: If any key Release technical personnel are requested to participate in the

strategic planning process, their functional manager will notify the Project Manager immediately. The functional manager will inform the Project Manager of what impact the Release will suffer as a result of the participation of the individual and the Project Manager will report such information to the DAD GM. Every effort will be made to find a substitute for the key person, either for the strategic planning or for the Release. No schedule slippage of the Release will be allowed without the approval of the DAD GM.

9.4 Field Application Engineering (FAE) Training

Impact: High
Probability: Medium
Triggering Event: FAE Training, week of July 15

Action Plan: The Project Manager will give the Training Department a list of those managers and engineers assigned to the project and will request that the Training Department not schedule any of those people as instructors. If the Training Department needs an individual already on the project, the Training Department will submit a written request to the individual's manager with a copy to the Release Project Manager. The individual's manager will attempt to locate an alternate instructor. If one is not available, the Training Department will eliminate that particular course from the curriculum and make arrangements to have the training done after the Release is completed. Such training could be via video tape or done at regional meetings in the field.

REFERENCES

Ahuja, H.N.; Project Management: Techniques in Planning and Controlling Construction Projects, 1984, John Wiley & Sons, New York, N.Y.

Ashley, D.B. & Avots, I.; Influence Diagramming For Analysis of Project Risks, Project Management Journal, March, 1984.

Cleland, D.I. & King, W.R.; Systems Analysis and Project Management, 1983, McGraw-Hill, New York, N.Y.

Collier, C.A. & Ledbetter, W.B.; Engineering Cost Analysis, 1982, Harper & Row, New YorK, N. Y.

Computer Associates; Super Project Expert Software Documentation1988, San Hose, CA.

Gray, C.F.; Essentials of Project Management, 1981, Petrocelli Books, Inc. OSU.

Hosley, W.N.; Innovative Approaches To Quality Assurance In Project Management, Project Management Journal, June, 1985.

Ireland, R.L. & Shirley, V.D.; Measuring Risk in the Project Environment, The Project Management Institute Symposium Proceedings, Sept., 1986.

Jelen, F.C.; Cost and Optimization Engineering, 1970, McGraw-Hill, New York, N.Y.

Keen, J.S.; Managing Systems Development, 1981, John Wiley & Sons, New York, N.Y.

Kepner, C.H. & Tregoe, B.B.; The Rational Manager , 1965, Mcgraw-Hill, New York, N.Y. (Note: a later edition, the New Rational Manager, is also on the market, but was not available to the author.)

Luengen, J.L.; Project Management Workshop Manual, 1988, Vancouver, WA.

Murray, J.W. & Ramsaur, W.F.; Project Reserves A Key to Managing Cost Risks, Project Management Quarterly, Sept. 1983.

Silverman, M.; The Art of Managing Technical Projects, 1987, Prentice-Hall, New Jersey.

Traylor, R.C., Stinson, R.C., Madsen,J.L., Bell, R.S., & Brown, K.R.; Project Management Under Uncertainty, Project Management Journal, March, 1984.

Wideman, R.M.; Risk Management, The Project Management Journal, Sept., 1986.

CONTROL CHARTING IN SOFTWARE DEVELOPMENT

Lionel L. Craddock
IBM Application Business Systems
Hwy 52 & 37th St NW
Rochester MN 55901

Abstract:

Control (Shewhart) charts are commonly used with industrial processes to ensure that process output remains within the limits specified. They effectively direct attention to special causes of variability in the process, and they can indicate the amount of common variability that must be reduced by management action.

This paper describes an application of control charts in the development of a large systems software project. It discusses how metrics gathered during the inspection and testing stages are used to monitor the quality of specific parts of the system. If required, corrective action can be taken at an early date. The goal of the effort is to improve the final software quality by providing developers and management with an early indication of which areas of the software may be defect-prone.

Biographical Sketch:

Lionel is manager of a data management development department, IBM Application Business Systems, Rochester MN. He has participated in process definition and metrics aspects of systems software development in Rochester.

He has developed design automation and systems software for Texas Instruments in Dallas and Austin. He managed System Test and Software Quality Assurance organizations for Data Systems Group, Texas Instruments, Austin. Lionel has taught Computer Science and Physics classes at West Virginia Wesleyan College, Buckhannon WV; and he has been an applied researcher at West Virginia University, Morgantown WV.

Introduction

This paper discusses a software quality control technique, control charting, that has been around for many years and used extensively in manufacturing. The paper presents some examples of applying the technique to the inspection and testing stages of the software development process. This paper does not address the question of whether or not the development process is in a state of "statistical control".

There are two types of variability in a process. A controlled or common cause of variation is one of the many random sources of variation in a process that is in statistical control. An uncontrolled or special cause of variation is not random or natural to the process and can lead to unpredictable output. Walter Shewhart of Bell Laboratories developed control charts, a simple but powerful tool, to determine whether a process is operating in a state of statistical control and to identify uncontrolled variation in a process. Deming (see Reference 2, Chapter 11) is an excellent introduction to these common and special types of variability.

Control charts are routinely used in manufacturing processes in the United States and other countries, notably Japan, to direct attention to special causes of variation. Their application to software development is more recent.

There are different types of control charts (3). In this study we use the most common type of the variables charts, the 'X bar' chart, to identify aspects of the development process that are 'out of control'. That is, we attempt to point out portions of the system software that are potentially defect-prone so that corrective action can be taken as soon as possible.

A control chart is developed as follows:

1. Gather the data
2. Compute mean, standard deviation, and control limits
3. Plot the data

The control limits are plus and minus three standard deviations from the mean. In a sample from a normal distribution, an observation would occur outside the control limits by chance (common cause) less than 0.3 percent of the time. Some of the measurements we have studied include inspection defects, inspection effort (preparation and meeting time), rework, and test defects. We have taken various data from an ongoing development process, plotted control charts for many of the observations, and fed back the information to developers and management to help

identify defect-prone areas in the software. Recovery actions, whether during the inspections or testing, include redesigns and reinspections, adding more resource to the component, strengthening the test scenarios, and changing the schedules.

Control charting is very effective for isolating potentially defect-prone pieces of a software product for management action. It is easy to point to the item that is outside the control limit and ask for the explanation as to why it is not defect-prone or, if it is believed to be a quality problem, what the plan is to fix the problem.

Another use of the control charts is to show a trend in the overall process. For example, if the inspection packages are becoming larger, management can try to determine if the process is becoming more efficient or if this trend is due to increasing schedule pressure. Another example: if the inspection preparation effort over time is decreasing, it could indicate that the programmers are becoming more familiar with the project and process or that there is less available time to prepare.

As an introduction to these types of charts, we take an illustration from Deming (2, p. 310). Figure 1 is an example of a run chart. It gives the miles per gallon from one fill-up to the next over a period of months, the average of 25 miles per gallon having already been established. It shows the mileage at or below the average for nine successive fill-ups, indicating a special cause of variation. A few successive points below or above the average can be expected, but seven usually indicates a special cause. The chart does not indicate what the cause is; possibilities include stop-and-go traffic, mountain driving, different brand of fuel, bad spark plugs, and so on, including combinations of causes. A number of causes were ruled out, leaving spark plugs as the primary explanation. New spark plugs restored the performance to expected levels.

The remaining sections of this paper are organized as follows:

First, we present information about the project. This includes a brief description of the system, highlights of the development process, and information on the data collection activities.

Next, we discuss the control charting aspects of the project, the feedback to developers and management, and the quality improvement actions taken in specific instances.

In conclusion, we list the actual and potential benefits of using control charts in software development.

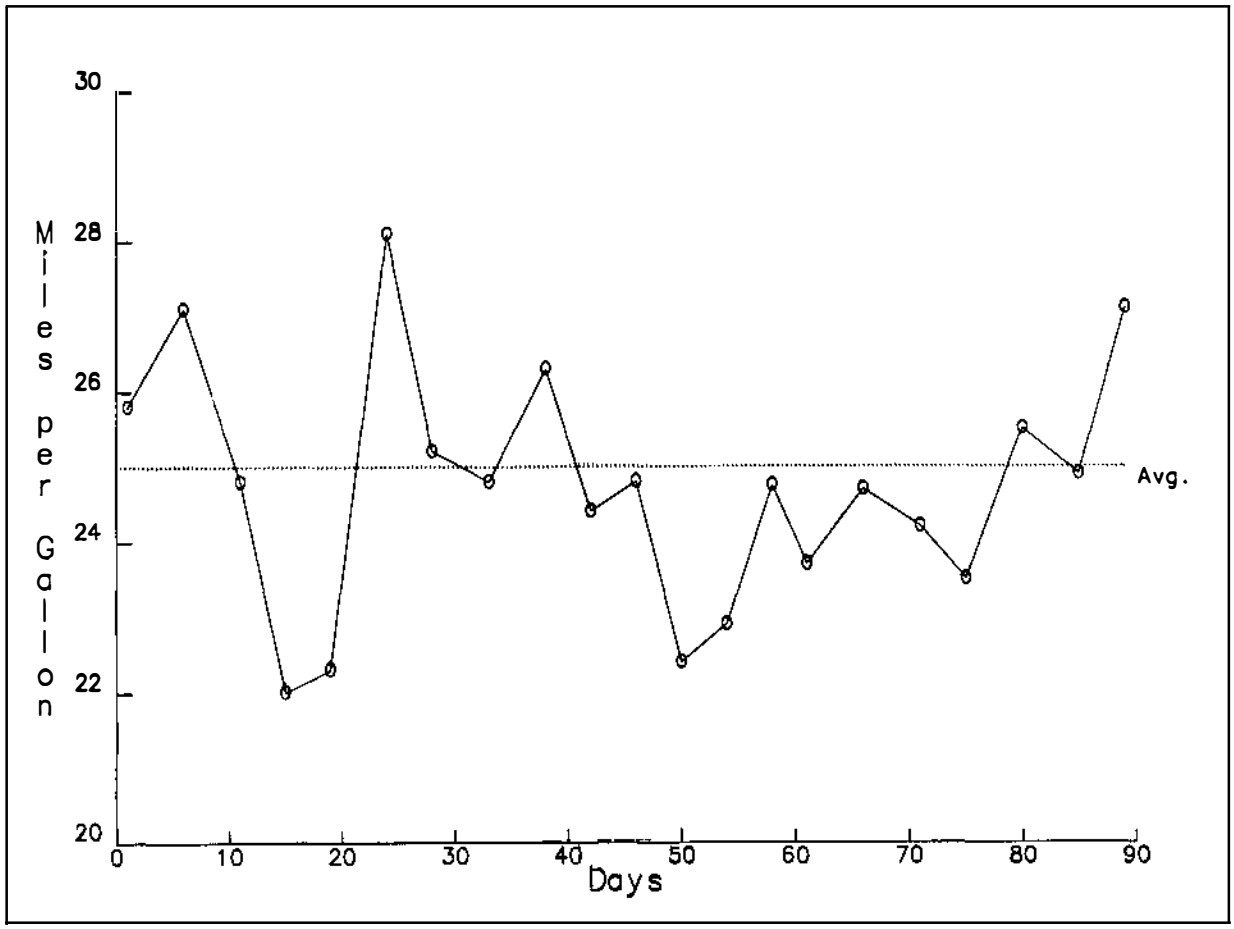


Figure 1. Sample chart showing a 'run' between days 42 and 75. New spark plugs after day 75 returned the miles per gallon to previous levels.

Project Overview

The System

The software development project described in this paper represents the initial release of a computer operating system. While a significant portion of the system consists of reused code, this study focuses on the more than two million lines of new and modified source code. The large majority of the software is written in high level, PL/1-like languages. The development effort has involved hundreds of programmers for more than two years.

The software development organizations are managed primarily along functional lines: data communications, data management, and integrated workstation/office. In the text and figures we refer to the major development organizations as Areas A through D.

Development Process

Although the different programming organizations work on different products and at different levels of the system, we have attempted to follow the same high-level development process. This process follows the software development life cycle model (1) and is similar to that described in IBM's *Programming Process Architecture* (4).

The inspection stages consist of the following:

I0: High Level Design Inspection
I1: Low Level Design Inspection
I2: Code Inspection

Inspections are also held for test plans and test cases, but in this report we consider only the product inspections.

The I0 inspections cover the intra-component designs, program control, and interfaces between modules or parts. The I1s ensure the designs of individual modules or parts and validate the transition from design language to implementation language. Finally, the I2 inspections cover the actual code implementation of the individual parts.

The inspection process is flexible. For example, software components may have combined high level design inspections or a single component may have multiple I0s. In some cases, if the high level design is considered moderate or low complexity or the implementation straightforward, the I1 inspection may not be held.

Software testing consists of the following sequential stages:

UT: Unit Test
CT: Component Test
PT: Product Test
ST: System Test

Unit testing occurs following the I2 inspection and involves exercising all normal and error paths in a single module. Unit testing is normally performed by the author of the module or part. Component testing consists of exercising intercomponent interfaces and functions after the modules that make up the components have been integrated into the product library. This testing is performed by the groups of developers responsible for the components or by a component test team that reports to the development organization. Product testing combines the components that make up products or significant functional pieces of the system. This test stage is the responsibility of an independent test group. System test, the responsibility of an independent test group, is the testing of all the products and associated hardware that make up the system for the release.

Note: Product testing is functionally the same as system testing but occurs before the entire system is available.

Pervasive issues, such as performance, usability, and reliability, are important items throughout the development process. Guidelines and checkpoints help monitor these items during the inspections as well as during the various testing stages.

Data Collection

Although much information is recorded for the inspections and testing activities, we will, in general, discuss only those that figure into the control chart examples.

Inspection Metrics

Data collection began with the initial high level design inspections. The control charting began about eight months into the project when there were enough inspections for the individual organizations to analyze.

Inspection metrics include cost-of-quality metrics such as preparation hours, meeting hours, and rework hours. Our size metric is lines of executable product code (LOC). We record the actual LOC on a per-part basis for the I2 inspections; estimates are recorded for the other inspection types (per part for the I1s). Non-commentary source instructions, including declarations, make up the LOC counts; compiler directives are not included. Figure 2 summarizes the guidelines for counting lines of code.

HIGH LEVEL LANGUAGE

- One LOC for each statement delimited by a semicolon.
- One LOC for each

  variable
  macro
  IF
  THEN
  ELSE
  message

  Excludes compiler directives.

ASSEMBLY LANGUAGE

- One LOC for each non-commentary source statement.

  Includes executable and non-executable statements.
  Excludes assembler directives.

Figure 2. Guidelines for Counting Lines of Code

Defects are tabulated for each inspection, on a per-part basis for the I1 and I2 inspections. Defects are inspection items that

1. If uncorrected, could result in customer-reported problems, or
2. If uncorrected, could result in conditions in a later inspection package or test stage that are defects, or
3. Do not conform to documented specifications or requirements, for example, performance, usability, and so on.

Although additional defect information is collected, such as type of defect and whether or not the defect should have been discovered at a previous inspection, only counts of defects are pertinent to this study.

Figure 3 is a sample of a report based on the inspection data. Figure 4 is an inspection summary for a few of the components; it shows defect and effort data aggregated by type of inspection.

Insp ID	Insp Type	Re-In?	LOC	Defs	Defs /KLOC	Prep Hours	Meet Hours	Total Hours	Re-work	Rewrk /KLOC
CRR003	I0	N	415	4	9.6	3.5	0.0	3.5	0.0	0.0
DRW001	I0	N	500	4	8.0	4.0	1.5	5.5	0.0	0.0
JSE001	I0	N	800	3	3.7	5.0	4.0	9.0	0.0	0.0
PAC015	I0	N	4000	42	10.5	5.0	25.1	30.1	60.0	15.0
LWK004	I1	N	100	2	20.0	0.5	1.0	1.5	0.7	7.0
LWK002	I1	N	190	5	26.3	0.0	3.5	3.5	2.0	10.5
LWK005	I2	N	100	4	40.0	0.5	1.0	1.5	0.8	8.0
LWK003	I2	N	190	1	5.3	0.0	3.5	3.5	1.5	7.9
DWJ006	I2	N	2340	16	6.8	15.7	18.8	34.5	8.0	3.4
			8635	81		34.2	58.4	92.6	73.0	

Figure 3. Example inspection data report

Comp	Insp Type	#	LOC	Dfts	Dfts /KLOC	Pr Hr /KLOC	Mt Hr /KLOC	Tot Hours	Tot Hr /KLOC	Re-work	Rwork /KLOC
D2	I0	16	4352	123	28.3	21.6	30.3	225.8	51.9	18.8	4.3
D2	I2	22	5281	37	7.0	1.5	12.2	72.6	13.7	11.3	2.1
		38		160	35.3	23.1	42.5	298.4	65.6	30.1	6.5
D3	I0	1	2850	36	12.6	1.6	4.2	16.4	5.8	3.0	1.1
D3	I1	1	3750	3	0.8	0.3	0.8	4.0	1.1	2.5	0.7
D3	I2	1	3202	10	3.1	0.5	0.9	4.5	1.4	2.5	0.8
		3		49	16.6	2.3	5.9	24.9	8.2	8.0	2.5
W6	I0	12	3070	54	17.6	19.1	19.3	118.0	38.4	21.6	7.0
W6	I1	10	3687	51	13.8	16.2	9.1	93.3	25.3	48.0	13.0
W6	I2	20	3242	91	28.1	27.6	22.7	162.9	50.2	37.7	11.6
		42		196	59.5	62.9	51.1	374.2	114.0	107.3	31.7
D1	I0	7	3150	71	22.5	15.8	21.4	117.1	37.2	53.1	16.9
D1	I1	5	3340	76	22.8	17.2	14.0	104.4	31.3	29.0	8.7
D1	I2	12	2785	80	28.7	33.1	20.1	148.3	53.2	122.0	43.8
		24		227	74.0	66.1	55.6	369.8	121.7	204.1	69.3

Figure 4. Example inspection data summary by component and inspection type

Testing Metrics

Unit testing normally occurs prior to code integration. The only unit test metric applicable to this study is defect counts. Data reporting is the responsibility of the unit tester, usually the code developer.

Figure 5 shows a portion of a sample report based on the unit test data.

Id	Dept	Prod	Comp	Part Name	Part Type	Test Effort	Repair Effort	% LOC Tested	Total Dfcts
SJK001	481	PRD1	D4	XD4DSPOV	MOD				16
SJK002	481	PRD1	D4	XD4XXX	PLMI				0
SJK003	481	PRD1	D4	XD4COPEN	PLMI				1
SJK004	481	PRD1	D4	XD4ACXDU	PLMI				2
SKJ004	481	PRD1	D4	XD4ACXDU	PLMI				0
TRT001	48M	PRD1	D4	XD4GETST	PLMI	38.0	6.0	96.4	18
6	(#)		D4			38.0	6.0		37

Figure 5. Sample unit test data report

As the code is integrated it is placed under change control; the reporting and tracking of much of the test data is associated with this change control process. The PTR (Problem Tracking Report) is the vehicle for initiating a fix for a defect found in integrated software.

PTR metrics include cost-of-quality metrics such as reporting effort and answer effort. However, for quality control purposes, the PTR is the primary datum.

Note: The PTR contains a valid/invalid indicator for whether the report signifies a real defect or not, and PTRs can count as valid defects although no code changes occur as a result.

The LOC counts used in the control charting for the testing metrics are kept separately from the PTR data. The lines of code are counted at the initial integration and for each successive integration on a part basis.

Figure 6 is one example of a report based on the PTR data.

Comp-Prod	LOC	No. Opn	Open /KLOC	No. Ans	Answd /KLOC	No. Fix	Fixed /KLOC	No. Clsd	Closed /KLOC	Val. /KLOC
A9 -PRD2	3140	2	0.6	3	1.0	.	.	17	5.4	7.0
C1 -PRD1	1924	1	0.5	2	1.0	2	1.0	34	17.7	20.3
C2 -PRD1	11945	2	0.2	4	0.3	2	0.2	148	12.4	13.1
C3 -PRD1	2918	2	0.7	10	3.4	.	.	58	19.9	24.0
D3 -PRD1	57570	18	0.3	20	0.3	9	0.2	301	5.2	6.0
D4 -PRD2	12900	.	.	3	0.2	1	0.1	14	1.1	1.4
E2 -PRD1	1500	1	0.7	3	2.0	8	5.3	31	20.7	28.7
E8 -PRD1	10670	2	0.2	6	0.6	4	0.4	64	6.0	7.1
K1 -PRD1	676	.	.	.	.	.	.	15	22.2	22.2
L4 -PRD1	4790	2	0.4	2	0.4	.	.	75	15.7	16.5
L5 -PRD2	3750	.	.	.	.	.	.	31	8.3	8.3
M3 -PRD1	5558	12	2.2	8	1.4	8	1.4	86	15.5	20.5
M6 -PRD2	681	.	.	.	.	.	.	8	11.7	11.7
N2 -PRD1	4013	.	.	3	0.7	7	1.7	105	26.2	28.7
P3 -PRD1	11705	6	0.5	7	0.6	.	.	46	3.9	5.0
P5 -PRD1	9105	6	0.7	.	.	1	0.1	26	2.9	3.6
P6 -PRD1	3400	9	2.6	5	1.5	4	1.2	48	14.1	19.4
R3 -PRD2	19712	2	0.1	4	0.2	4	0.2	55	2.8	3.3

Figure 6. Example PTR data report

Control Chart Examples and Usage

The control charting activities began early in the development cycle, after enough inspections had occurred to give the charts some validity and significance to the developers and managers. As test defect counts became available and the inspection activity diminished, the PTR data became the focus of the charting.

The information was presented periodically in open meetings to the development personnel, both programmers and managers, along with other project quality and status information. The management staff of the development organizations received monthly summaries of the control chart studies.

Note: The control charting work was performed using an in-house statistical and graphics package that was written primarily in APL.

Examples

Here we present several samples of control charts produced during system development. These represent a fraction of the metrics that were investigated, the focus being on those types of charts that were perceived as most useful.

Figure 7 is a plot of defects per KLOC (thousands of lines of code) for a series of I2 inspections over time. This type of chart can help identify portions of the software that are likely candidates for reinspection, redesign, and so on. It can also show potential trends. For example, if the defects/KLOC counts over time are trending downward, this may indicate that the inspection process is getting less emphasis than needed.

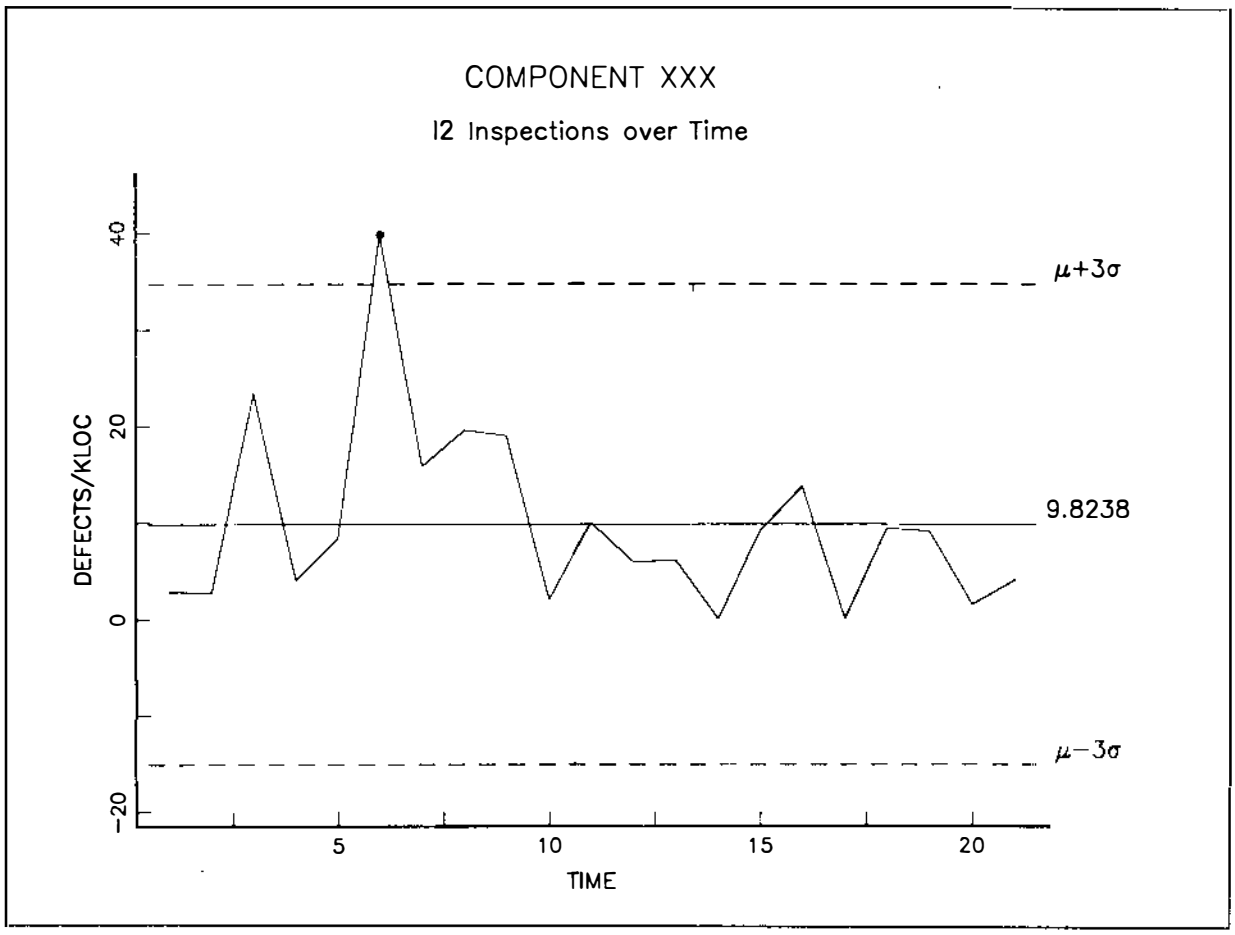


Figure 7. I2 defects/KLOC over time

Plotting the cost-of-quality inspection metrics over time can also show potentially troublesome trends such as inspectors spending less time preparing for the inspections, exceeding the project guidelines for LOC inspected per meeting hour, and so on.

Figure 8 shows rework hours per KLOC for a series of the system components. At the end of an inspection, the author(s) of the material must repair or answer all the defects. The rework is a measure of this repair effort. The premise here is that a component (or inspection) that requires extraordinarily high rework is also a candidate for some additional inspections or some other recovery actions. (As one would expect, we have found that rework effort has a high positive correlation with defect counts.)

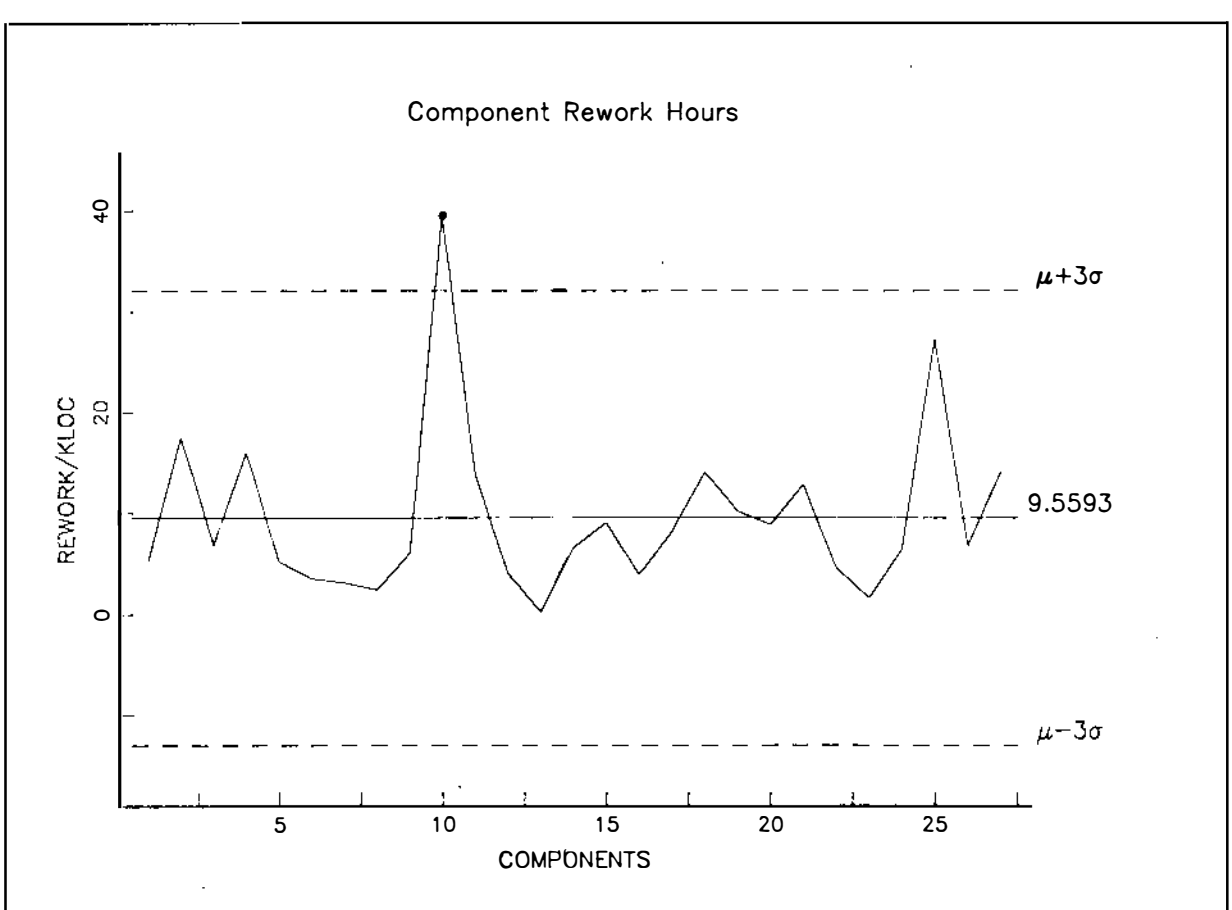


Figure 8. Inspection rework/KLOC by component

Probably the most studied and most frequently plotted of the inspection metrics is the total inspection defects per KLOC. This is the sum of the defects/KLOC across inspection type for each component, 'Dfts/KLOC' column in Figure 4. The four control charts in Figure 9 show the total inspection defects/KLOC by component for the development organizations.

During the testing stages, management focused on the PTRs, getting the defects fixed, the fixes integrated and tested, and so on. In our control charting, the focus was again on total defects and total defects/KLOC. Note that for the inspection metrics the normalizing KLOC is the inspected lines of product code, while for the PTRs it is the integrated LOC. The four control charts in Figure 10 show the total test defects (PTRs/KLOC) by component for the development organizations.

We explain to the viewers of these charts that if the population from which the data samples were taken were distributed normally, then a point outside the control limits would occur by chance less than three times in one thousand. Thus in most cases there is a special cause for the observation lying outside the limits, one that is likely related to quality. It is then up to the owner of the software behind that observation to investigate the cause and, if a quality problem exists, fix it. (In general, the isolation and removal of special causes of variability are the responsibility of someone directly connected with the process (2, p. 320).)

Note: It is obvious from the example charts that the data are not from normal distributions; we explain this fact to the viewers of the charts, but state that even though the by-chance occurrence of a point outside the limits is higher, the information is still useful and can help isolate and repair defect-prone areas of the project.

Usage

We periodically presented to the programming community the various control charts on the defect data, cost-of-quality data, and so on. Generally, the owning manager of the portion of the project that was shown to be outside the control limits was required to explain the situation. If a quality problem existed, an action plan was formulated. Actions in particular cases included reinspections, redesigns, adding more expert resource to the development effort, and adding more test resource.

Sometimes the charts would just point out an error in the data, for example, the LOC number was misrepresented. Certainly, if the data were deficient in some

manner the charts can be used to encourage its correction, for example, by asking the inspection author and moderator to complete and report all the inspection work.

In many instances, when the control charts were presented, the management personnel responsible for a particular portion of the project that was outside the control limits would respond that he/she was not too surprised, that is, the graphs were confirmation of his/her suspicions. He/she would generally have other evidence to substantiate the premise that there really was a quality problem.

Although some control charts were shown to the management staff of the development organizations each month, we were at times asked to produce additional charts for specific metrics intermediate to these meetings to see if a particular situation had improved or just for additional information. Lower level managers would request specific control charts that they would present to higher management, often as evidence that their products are in control or to demonstrate that actions they have planned are really necessary.

Several of the managers were not content with seeing just the components that were outside the control limits. They requested that the components that were outside (or very close to) the control limits be removed and the data plotted again. This process of 'peeling the onion' permitted the identification of the next set of potentially defect-prone components, some of which may have been masked on the initial set of charts. The charts in Figure 11 show the total test defects (PTRs/KLOC) by component for the development organizations after the initial set of points outside the control limits have been removed. Figure 11 uses the same set of base data as Figure 10, but with four points removed for the Area A organization, one for the Area B organization, two for the Area C organization, and two points removed for the Area D organization. This process of 'peeling the onion' can continue for any number of iterations.

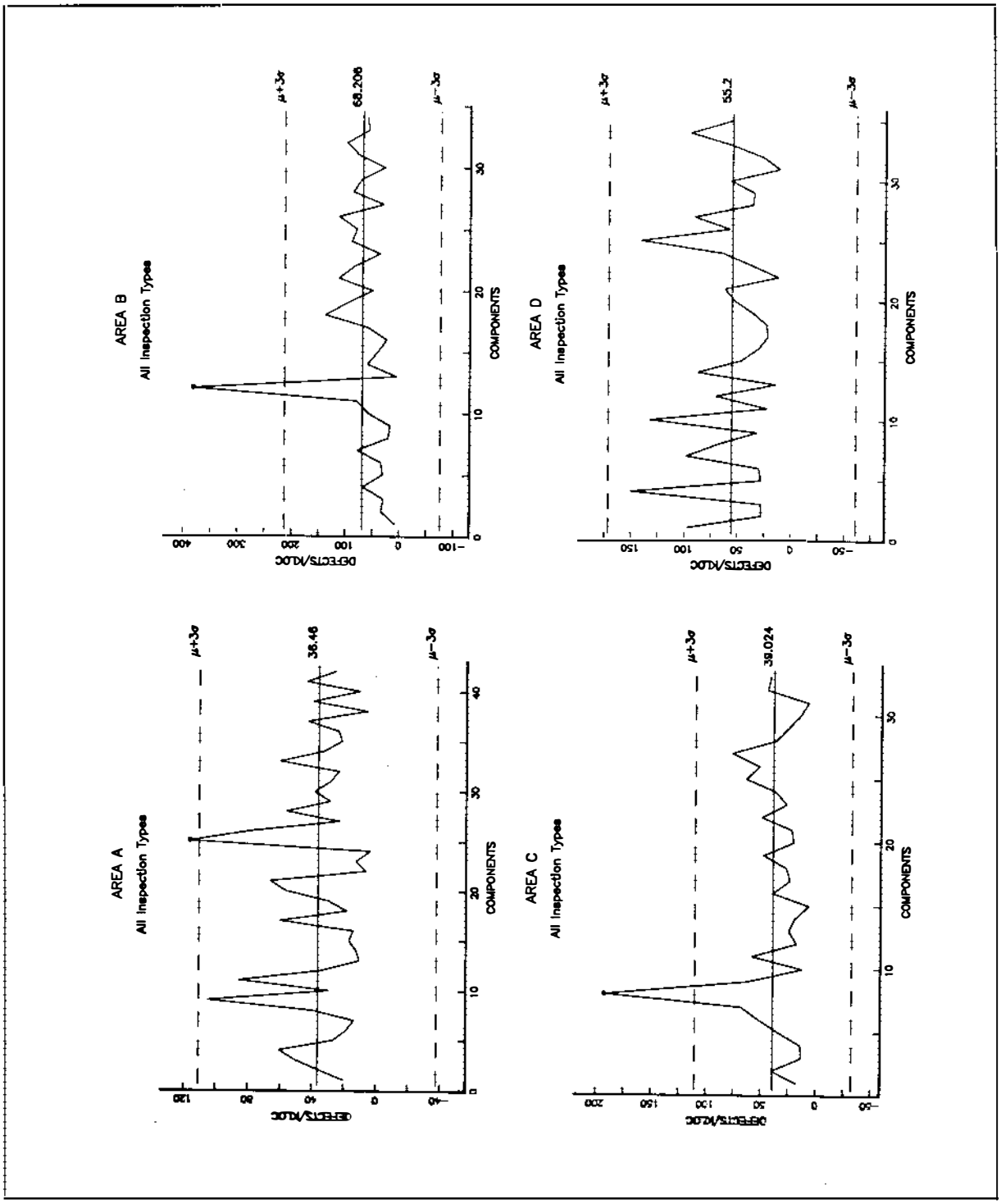


Figure 9. Total inspection defects/KLOC by component for each organization

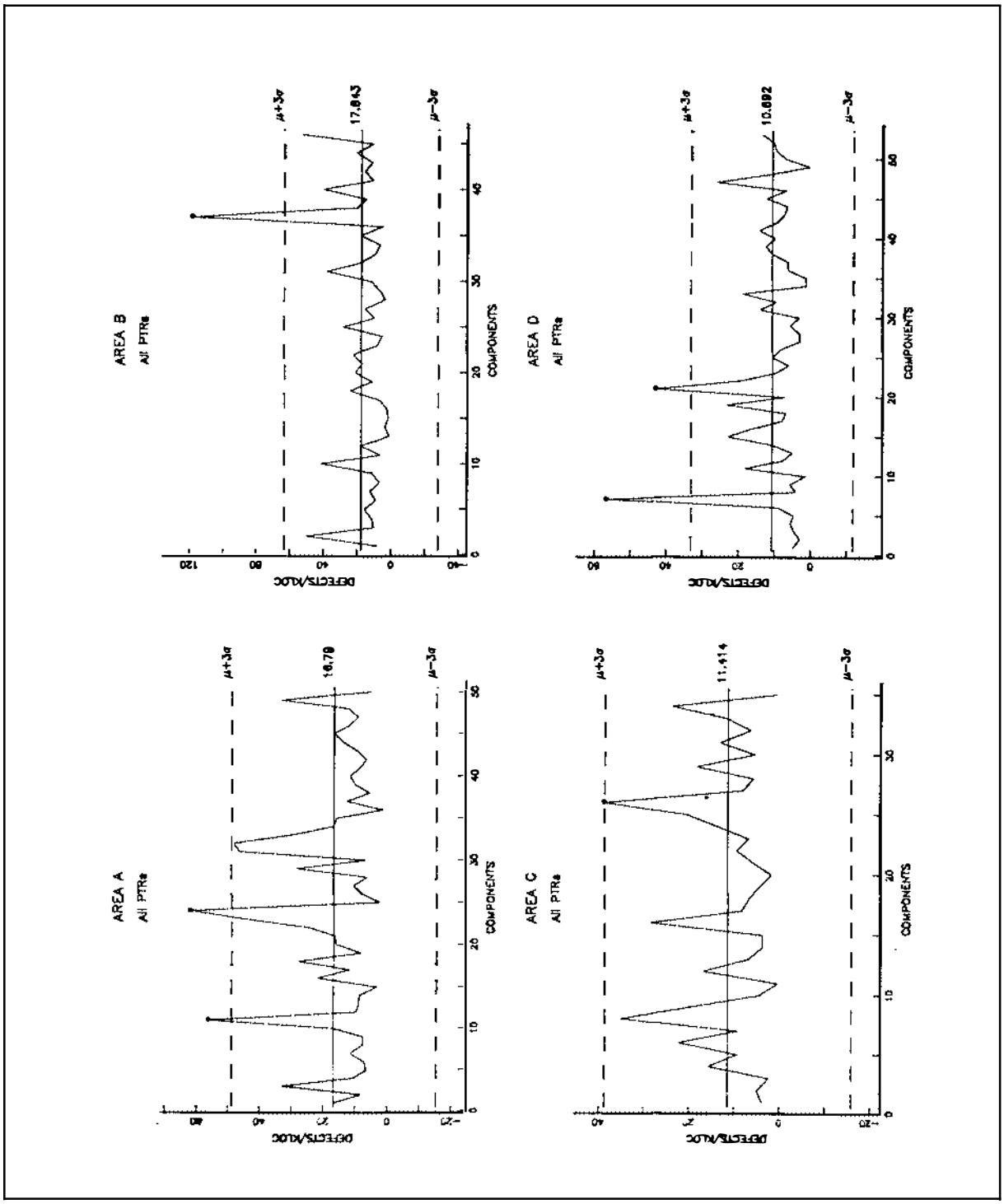


Figure 10. Total test defects (PTRs/KLOC) by component for each organization

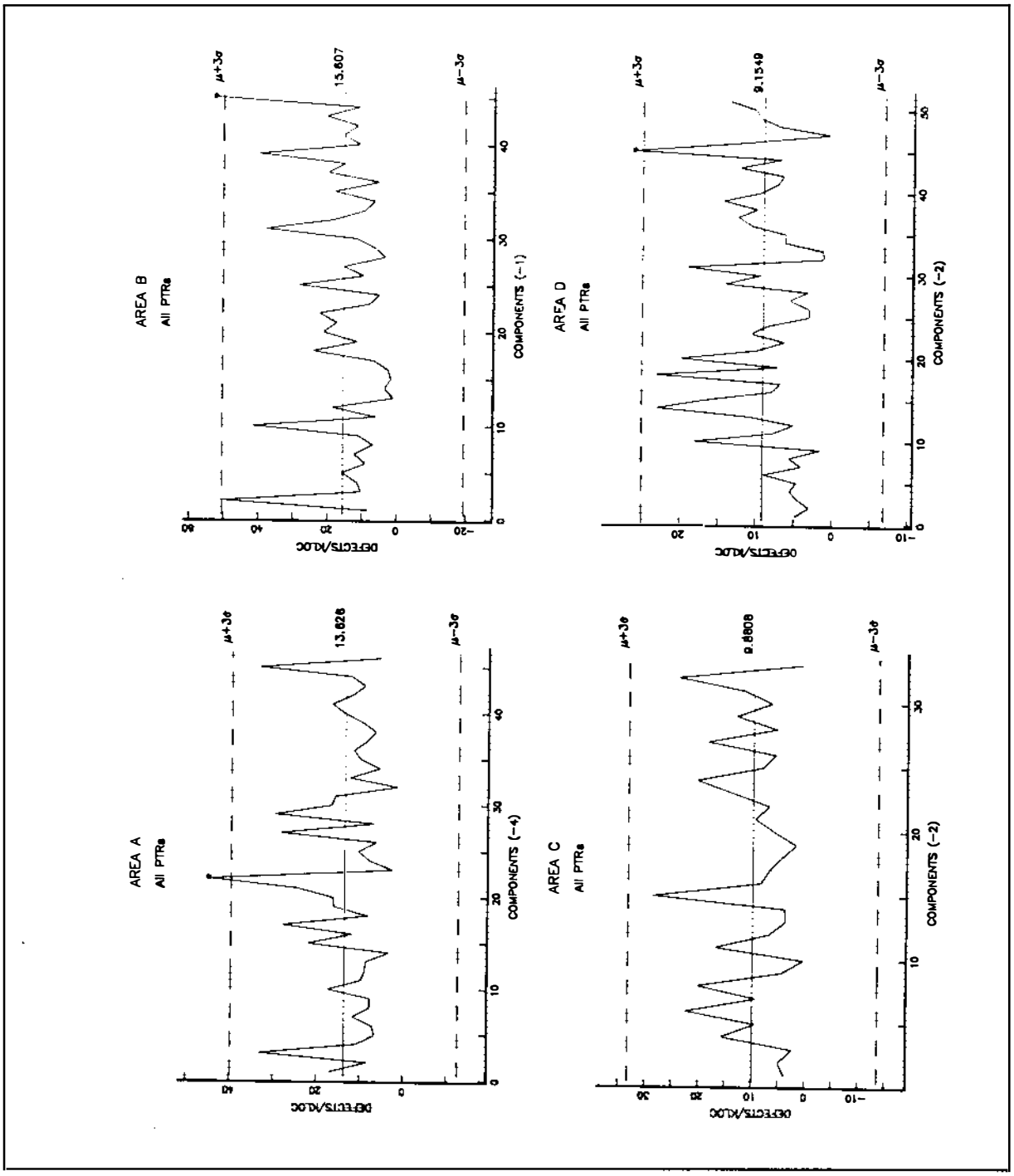


Figure 11. Test defects (PTRs/KLOC) by component for each organization. These charts show the data after the initial set of components outside the control limits (or nearly so) have been removed.

Conclusion

The project discussed in this paper had many processes and procedures in addition to control charting to focus on quality deliverables, so it is not possible to assess the direct impact that this effort may have had. The reinspections, additional design work, added test resource, and so on, may have been initiated without producing the control charts. However, it was obvious that the development community did take an interest in and did use these charts.

We judge the most important aspect of the charts to be the visual impact to management. The potentially offending piece of the project stands out on the graph. The owner must then explain that it is there just by chance (highly unlikely); that yes, it was a quality problem, but it is now corrected (or an action plan is in place to correct it); or that it is due to a special cause that is not a quality problem. (One example of the latter is the situation in which the software being reported on has a very low LOC count, so that a few minor defects can produce a high defects/KLOC number.)

We have discussed control charts and presented examples of how they can be used to improve software quality, primarily by allowing the developers and management to focus resource where the potential for return is greatest. We recommend that any development organization of significant size include the use of this technique in their procedures.

Acknowledgements

I thank Dan Stott of IBM in Kingston NY for an introduction to control charting. I thank Greg Boone, Gene Leitner, and Miller Ness for their reviews of and comments on this paper.

References

1. Aron, J. D., *The Program Development Process, Part II The Programming Team*, Van Nostrand Reinhold Company Inc., New York, 1984.

2. Deming, W. Edwards, *Out of the Crisis*, Massachusetts Institute of Technology, Cambridge MA, 1982.

3. *Process Control, Capability and Improvement*, International Business Machines Corporation, Southbury, Connecticut, 1985.

4. *Programming Process Architecture*, Version 2.0, IBM IS&SG, Poughkeepsie NY, April 1986.

FIVE COMPONENTS OF A SOFTWARE QUALITY ASSURANCE PARADIGM

Patrick H. Loy
The Johns Hopkins University

ABSTRACT

The current paradigm for software quality assurance (SQA) is a vast improvement over that of earlier periods. However, it still needs improving, especially with respect to integration and scope. In terms of integration, SQA must become an integral part of every task in the development process, rather than being a parallel function embodied in a separate department. With respect to scope, an effective SQA model must encompass areas that are often seen as marginal to software quality. In particular, the areas of business policy, metrics, peopleware, technology transfer, and technical leadership must be addressed.

BIOGRAPHICAL SKETCH

Patrick H. Loy serves part-time on the faculty of The Johns Hopkins University teaching graduate courses on software engineering topics. He also owns his own consulting firm, and has extensive experience in the areas of software design, testing, and quality assurance. His recent activities include conducting an evaluation of software testing practices at the Social Security Administration.

Mr. Loy has taught software testing and quality assurance seminars for numerous organizations throughout the U.S. and Canada, and is currently involved in training software acquisition managers at the National Security Agency. His career includes having been a network software analyst for the Control Data Corporation Cybernet System.

Mr. Loy has refereed technical manuscripts for the Addison-Wesley Publishing Company, and for various professional conferences. He is a member of the IEEE, ACM, and Computer Professionals for Social Responsibility. Mr. Loy holds the M.S. degree in computer science from The Johns Hopkins University, and the B.S. degree in mathematics from the University of Oregon. He can be reached at:

Patrick H. Loy
Loy Consulting, Inc.
3553 Chesterfield Avenue
Baltimore, Maryland 21213
(301) 483-3532

FIVE COMPONENTS OF A SOFTWARE QUALITY ASSURANCE PARADIGM

Patrick H. Loy
The Johns Hopkins University

Introduction

Historically the concept of software quality assurance (SQA) has evolved from being merely a compliance check on the final product or deliverable (circa 1955) to being viewed today as a life cycle activity with input into all phases of software development. An artifact of this model is that many organizations have separate SQA departments,[1] some of which have relatively well-defined autonomous powers and authority.

While this paradigm for SQA is a tremendous improvement over that of earlier periods, it still needs improving. The problem is one of integration and scope. With respect to integration, the weakness of the current model is that it views SQA as a parallel process, an activity that takes place concurrently with the other activities in the development cycle, and which provides the necessary checks and balances on them. Ultimately, there should be no need for this kind of division of labor. The concepts and functions of SQA should be completely integrated into the development process, making an anachronism of the notion of a separate SQA department.

With respect to scope, the current model is weak in that it sees SQA as the responsibility of a subset of the organization, i.e., it is a compartmentalized function within the technical group. The scope of SQA should be viewed in the broadest of contexts which extends well beyond the confines of the technical groups. SQA, then, must be recognized as involving all organizational levels, from the board of directors (or other comparable body) on down.

This paper presents a discussion of five components that are often viewed as peripheral to the SQA function, but are central to a meaningful SQA paradigm. This discussion is meant to broaden the reader's view as to what constitutes the subject matter of SQA, and to lay the foundation for a more integrated approach to the matter as discussed above. The components are not meant to constitute a model themselves, nor is it assumed that there is one fixed universal paradigm. The intent is to focus attention on these matters, and their interconnectedness, and to stress their relationship to SQA.

Defining the basic concepts

Before discussing the five components, a word about definitions is necessary. One of the problems with the field of software engineering is the lack of standard definitions for the terms we use. The problem is exacerbated in the area of software quality assurance because the notion of "quality" itself is dependent on contextual interpretation. The IEEE definiton of "quality" illustrates the point: *the totality of features and characteristics of a product or service that bears on its ability to satisfy given needs.* [2] The "given needs" part of the definition allows for almost any interpretation. However, there are some attributes which most software people identify with quality, such as *reliable, maintainable, efficient, portable*, etc., and the relative emphasis placed on these attributes forms the basis of an intuitive notion of SQA. This paper will assume that each reader has his or her own intuitive definition, and will not attempt to constrain it in any way.

Five important components of the SQA paradigm

1. Business policy. In May of this year a three-day SQA workshop was held at the Johns Hopkins University, attended by SQA specialists from industry and government.[3] One session focused on the relationships between business policy and corporate values, and the production of quality software.

A consensus emerged that the principle contradiction in private industry in this respect is the short-term profit orientation. A manifestation of this is that the delivery date, and corresponding development schedule, become market-driven rather than product-driven. (There is a parallel dynamic in government organizations with political pressures, instead of profit, being the driving force.) One of the chief complaints expressed by the participants in the workshop was the "end of the life cycle crunch." Time estimates prove to be unrealistic, schedules slip, new requirements are imposed upon the developers during the project, and yet the delivery date remains constant. As a consequence, the testing phases, and other verification and validation activities held at the end of the development life cycle, are short-changed.

This problem is particularly acute in companies where the requirements for new products originate with the sales force. Typically, the salesperson promises the customer a particular product on a particular date, and the developers have little say in the matter. When the developers complain, management always sides with the salespeople who argue that "we will lose the account if we don't deliver this on-time." And that argument, of course, is tough to prevail against.

The second most important business policy contradiction that affects SQA, as identified by the workshop, is the lack of knowledge by top management as to

what software is all about. Management tends to not appreciate the complexity of software and the corresponding implications for the development cycle, and that the quality assurance activities for hardware are a very different matter than for software.

In part, this problem is a manifestation of the relative newness of software development as a special field. We've been building hardware-based systems for a long time, and it is not unusual for there to be former hardware engineers in top management positions. It *is* unusual, however, to find a *software* engineer in a top management positon. In fact, many companies that were hardware intensive as recently as five years ago are now software intensive. But top management has not changed, and much of middle management is people with many years of hardware training and experience who suddenly find themselves managing software development projects.

There are many more problems relating to software quality that are rooted in business policy and corporate values. The SQA workshop came up with a list of 25. However, it seemed that most of them were very closely related to the two mentioned above: short-term profit orientation, and ignorance on the part of top management as to "the nature of the beast. "

What can be done about these problems? The business policy problem is an area that software professionals frequently complain about, but usually feel powerless to do much about. From the list of 25 problems, the workshop participants were asked to identify the ones that "people like us" could have a direct impact upon. Originally only a few items on the list were checked. Through discussion, however, it became clear that technical people and QA managers could have at least some impact on every one of the items.

In fact, most of the participants had a story to tell as to how some concrete changes in business policy were made in a "bottom-up" manner, either in their own organizations, or ones they were familiar with. The key was in identifying what the contradictions are, and then brainstorming with colleagues to come up with a strategy to make change. An informal resolution was made that the conference participants would do this when they returned to their organizations.

The workshop group agreed that the toughest issue is that of short-term profit orientation. Top management, of course, must be very concerned about profit; they are, after all, accountable to the investors. And management itself is generally evaluated on the basis of attaining quarterly profit goals. Thus, it is not an easy contradiction to resolve. But, on the other hand, because they feel powerless to tackle the problem, most technical people have shied away from it. The sense of the group was that much could be done in this area if we start from

the premise that we can have an impact.

2. Metrics. It is an undeniable truism that a quality software product will not happen by chance, without careful planning and control being exerted by the project leadership - at least not on a consistent basis. Being "in control" is basic to reducing the impact of blind luck (good or bad), and for managing risk, which is perhaps *the* central goal of project leadership.

But what does it mean to be "in control?" At a minimum, it means that surprises are minimized during development, deviations are detected early, and the project lives up to its predictions. But for this state of affairs to be realized, feedback is essential.

The basic feedback mechanism is measurement. As Tom DeMarco says, "you can't control what you can't measure." [4] But, of course, measurement in itself is not enough. What to measure, how to gather the data to measure it, and how to use the results of the data collection are the big issues.

Many organizations seem to exhibit a degree of perplexity over the question of metrics. One reason for this is that they tend to approach the issue of *metrics* in the same manner that they approach the issue of *methods.* Hence, they seek to find the right metric that is out there in the field somewhere that will solve their problems, rather than recognizing that the question of developing useful metrics is largely a do-it-yourself matter. Organizations need to be committed to incorporating the basic metric function into their development life cycle, rather than to *finding* a set of specific metrics.

The basic metric function involves using metrics to assist in estimating, and to give guidance to the specific project phases. For estimating it includes the following steps: 1) making predictions prior to the start of the project (using the existing metrics); 2) gathering data during the project; 3) comparing the results of the data with the estimates after the completion of the project; 4) and then implementing the lessons learned from the experience by refining the metrics themselves, and refining the development procedures. The refined metrics are then used to help estimate on the next project.

For use as a development phase guide, a metric is used to assist in a specific development task. For example, the cyclomatic complexity metric[5] might be used to determine the appropriateness of a design partitioning, and to make adjustments as necessary.

The SQA workshop cited earlier spent a full day on the issue of metrics. Most of the organizations represented had little or no formalized metric efforts underway,

and were not capable of generating any useful metric relationships concerning defects, e.g., rate of defects in delivered code, cost to find and fix defects, relationship between defects and specific development approaches, etc. However, two companies were represented that do have metric groups that can produce such data, and they were swamped with questions from the rest of the workshop.

The workshop concensus was that defect measurement, or what DeMarco calls "quality accounting,"[6] was the most important metrics issue, the key questions being: who shall collect data about defects, and how; what protocol shall govern the use of defect data; what data shall be collected about each defect; and how shall defect responsibility be determined (some participants felt strongly that defect responsibility should not be logged against individuals at all). However, from the workshop as well as my own consulting experience it seems clear that there are few organizations who implement this type of quality accounting.

One of the main reasons why more organizations do not have a coherent metrics program is that management has not made the commitment to simply gather data for the long-range good of the organization. DeMarco speaks of the "unforgivable failure" of software project management: the failure to learn from past failures.[7] Some of the Japanese software houses have greatly improved their software quality over the past two decades or so, in large part due to their commitment to gather data on the current project so as to not repeat the same mistakes on the next project.[8]

3. Peopleware. The inspiration for the title of this section is from a book of the same name by Tom DeMarco and Timothy Lister.[9] Reading the book I was constantly impressed at how so many points "rang true" to my own experience. After some reflection, it struck me as to how central the topic is to an SQA paradigm.

People create software. Not methods, not tools, not techniques - they only help. Moreover, the costs of developing software are largely people costs. The central role that people, and their interactions with each other, play in developing quality software is not a new revelation. But the full implications of this reality are far from being recognized in our field.

Creating a quality product is not so much a technological as a sociological and psychological problem. Central to the issue are attitudes, work environment, the structure and organization of project teams, and the allocation of goals. It is largely a matter of unleashing people to perform to their maximum capabilities.

Of the "peopleware principles" for quality software development, two seem critically important. First, a culture must be nurtured that encourages people to see their jobs as that of a software quality engineer, regardless of their specific function. This means that product quality must be given high visibility, and the incentive for doing quality work must be clear and unambiguous.

When I teach short courses on software quality issues, I sometimes come across as "preaching" when it comes to the matter of reviews, walkthroughs, and inspections. But the reason I believe so strongly in them is because of what they do for the *culture* of an organization. They provide high visibility for the efforts of individuals and project teams, and they say loud and clear, "the quality of your work *matters.*" Many organizations have found that an excellent way to orient new hires to the quality objectives of the organization is to have them sit-in on walkthroughs as observers. The meeting itself addresses the SQA standards and policies of the organization as the product is evaluated. I have seen newly hired employees come out of such meetings with bubbling enthusiasm for their jobs, realizing that sometime soon they will be sitting in a similar meeting as the product author, thereby giving them the opportunity to "show their stuff."

Secondly, the work environment is critical to obtaining a quality product. If people are crammed into small, noisy cubicles, where they are frequently interrupted, they will not be performing to their potential. And, just as important, they will not be happy. DeMarco and Lister estimate that replacing a person typically costs about five months of the annual employee cost for that position.[10] Add to that the fact that the reason many people leave their jobs in the first place (being unhappy in their work environment) has caused them to be minimally productive long before they leave, and you can begin to see the extent of the problem. In general, we are not taking care of our most important SQA resource - *people*.

Lately, it seems that there is a growing interest in this topic, and it is an encouraging sign. The SQA workshop participants had much to say on the subject, and there was unanimous support for the proposal that peopleware is a central component of the SQA paradigm. During a break at a short course I recently taught at a client site (on real-time software design) one of the students saw me with the *Peopleware* book, and we began discussing it. Soon a crowd formed. It seemed that although the topic of the book was not central to the formal subject matter of the course, everyone in the class realized it was central to their own work experience.

4. Technology Transfer. Getting the right technology into the right hands is one of the major challenges facing software developers. There is always a need to come up with more and better methods and tools, but what software organizations

most need is to be able to incorporate and assimilate what is already out there into their own environments. When the Software Engineering Institute was formed at Carnegie Mellon University, one of the premises of its charter was that there is a twenty year lag between the creation of a state-of-the-art concept, and the incorporation of that concept into the standard practices of the field. Part of SEI's charter is to speed up that process.

The problem is not a simple one, for sure, but is a solvable one. Organizations are not often set up to accomodate changing technology, especially *development* technology. The pressures associated with the existing projects is frequently too great to spend much time thinking about how things could be done better with new methods and tools. In fact, it seems that most organizations change only when they are forced to change, i.e., they experience software failures that attract negative attention.

The relationship to software quality seems obvious: other factors being equal, you will build a better product with modern state-of-the-art tools, than with shabby, outdated tools. This law applies to all areas of science and engineering.

What can be done about this? This area is very closely related to the business policy issue, already discussed, and is a test of the long-range outlook of the organization. An organization must recognize that it must be structured, and funded, in such a way as to accomodate change in the area of development technology. This should be seen as a given. A percentage of the organization's budget should be earmarked for technology transfer, and it should be clear that this budget item will always be there. No one will ever "arrive" when it comes to this area - our field is changing too dynamically.

The problem of technology transfer is not the same as the problem of training. Most organizations have a training department of some sort. The training that is offered is, by and large, for specific projects underway, and is usually initiated by a project leader requesting the training department to provide a certain course on a particular subject. While this type of training is necessary, it does not address the technology transfer problem. Keeping the organization up to date on development technology requires a much different kind of commitment.

If the organizational structure can be made to accomodate this type of change, this problem is a very solvable one. Given the right environment, people love to learn new things. It is not unusual for a person to state that his or her main consideration when considering a job move is the opportunity to learn and grow. That fact speaks well of the people who are in our field. Of course every organization contains those who resist change, but it also contains a great many progressive-minded people who form the pool of latent leaders that can help the

company strive to be in the forefront of its field.

5.Technical Leadership. *Leadership is the process of creating an environment in which people become empowered.* [11]

Having the project team be influenced by the "right" people is clearly important for building a quality software product. The "right" people are problem-solving leaders who empower others. They put a premium on *innovation - on doing things a better way.* They stress understanding the problem, and work hard at coming up with creative ways to assist in this effort. They concentrate on managing the flow of ideas among their colleagues, and people feel encouraged to freely express their thoughts in their presence. They constantly have a watchful eye on maintaining the quality of the project, and they are known for the pride they take in the quality of their work. People feel relaxed and motivated working with them, and like being on their team.

But where do you find such people? In many cases it may be a matter of discovering them among the existing work force and helping to unleash their potential. Anyone who has ever worked in industry has seen certain people come to be recognized as leaders, irrespective of their specific assigned roles or places in the organizational structure chart. These people perform an invaluable function in the organization, and their contributions need to be recognized and encouraged. These people are the "diamonds in the rough" who can form the basis of a quality-oriented team.

However, almost everyone has at least some leadership potential, and it should be a primary concern of management to establish the setting where this potential can flourish. This necessitates that the organizational model of leadership be that which can best be characterized as "organic." This model stresses "systems thinking," as opposed to assuming a linear relationship between events; it sees power in the relationship, rather than in the role; it tolerates change, ambiguity and risk; and it tries to actualize the potential of the individual.

For leadership of this type to flourish, the organizational policies and standards must be compatible with this kind of leadership model. A critical issue here is that the policies must clearly delineate the boundaries between organizational standards and individual creativity. The domain of standardization must be well-defined, and continually re-evaluated. In addition, the training that is provided must reinforce these policies, and provide the opportunity for latent leadership qualities to surface. [12]

(The matter is analogous to theatrical training. Actors need to be trained so that they understand the "rules of the game," the principles of good acting. But the

training is meant to *channel* creative energy, not to *suppress* it. The key to good training, theatrical or otherwise, is to find that delicate, proper balance between constraint and creativity.)

In summary, organizations need to see the investment in leadership development as an investment in SQA - as a central part of the strategy to improve product quality.

Conclusion

In endeavoring to develop and refine an SQA paradigm we must put more stress on issues that are often seen as marginal to software quality. The above five items represent a start in that direction. Although they do not necessarily make a complete list, they are essential components of an SQA paradigm. As such, they represent an effort to begin to identify the essential factors important to attaining the two objectives stated at the outset: to make SQA an integral part of the development process, and to expand the scope of SQA to include environmental aspects that are not directly involved in development, but have a major impact on the final product. Hopefully this paper will help to stimulate further research and discussion on the issue of creating a stronger, more effective paradigm for Software Quality Assurance.

Notes and Annotated references

1. According to a recent survey by the American Society of Quality Control, 74 percent of the 108 respondents (The questionnaire was mailed to1,636 firms) had separate SQA departments. For a summary of the findings of the survey see *IEEE Software*, May 1988, pp. 102-103. Copies of the complete report are available from the American Society of Quality Assurance.

2. "ANSI/IEEE Std 729-1983, Glossary of Software Engineering Terminology," published in *Software Engineering Standards*, IEEE, 1987.

3. The workshop included representatives from General Dynamics, Elecronic Data Systems, I.P. Sharp Associates, National Security Agency, Systems Dynamics, Intel, Hewlett-Packard, Social Security Administration, Automatic Data Processing, U.S. Army, and Coppers and Lybrand.

4. Tom DeMarco, *Controlling Software Projects*, Yourdon Press, 1982, p. 3.

5. McCabe, T. J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, December, 1976, pp. 308-320. This metric, which is based on the number of decision points in the code, has been found to be useful as a software design complexity criterion.

6. DeMarco, op. cit. pp. 211-214.

7. Ibid, p. 6.

8. An interesting survey of Japanese software development techniques can be found in: Tajima, D., and T. Matsubara, The Computer Software Industry in Japan," *Computer*, May, 1981, pp. 89-96.

   For a discussion of defect metrics and data collection in Japanese software firms see: Abe, J., et al., "An Analysis of Software Project Failure," *Proceedings of the 4th International Conference on Software Engineering*, IEEE, 1979.

9. Tom DeMarco and Timothy Lister, *Peopleware: Productive Projects and Teams*, Dorset House Publishing Co., 1987.

10. Ibid, p. 105-108. *Peopleware* contains some very interesting and revealing statistics on the real costs involved in ergonomics, phone interruptions, employee turnover, etc.

11. Gerald M. Weinberg, *Becoming a Technical Leader*, Dorset House, 1986, p. 12.

12. In the view of this author, most organizations stress technical skills training to the detriment of leadership training, when in fact the emphasis should be the opposite.

Project Management 2.

"Information Scope and the OVERSEE Configuration Management Environment
Steven Wartik, University of Virginia

"Software Construction Project Management"
Robert E. Shelton, CASEware, Inc.

"Structured Computer Project Management"
William H. Roetzheim, The MITRE Corporation

INFORMATION SCOPE AND THE OVERSEE CONFIGURATION MANAGEMENT ENVIRONMENT

Steven Wartik
Department of Computer Science
Thornton Hall
University of Virginia
Charlottesville, Virginia 22903

ABSTRACT

This paper discusses "information scope", a simple but effective means to assess the quality of interaction mechanisms provided by a software development environment. Scope is a technique to categorize the types of interactions between project personnel, and the means by which tools support them. Scope is defined, and its influence on software development environments is covered. A taxonomy of tools is presented that follows from scope. The principles of scope are then applied through a discussion of OVERSEE, a software configuration management environment that has been greatly influenced by scope considerations. An experiment with OVERSEE is briefly discussed.

BIOGRAPHICAL SKETCH

Steven Wartik is currently an Assistant Professor of Computer Science at the University of Virginia. He also consults for the Software Productivity Consortium. Before coming to the University of Virginia, he was employed by TRW, Inc. in Redondo Beach, California, where he helped design and implement tools and techniques to improve software productivity.

His major research area is software development environments. He is interested in measures, both quantitative and qualitative, that will help software personnel understand the applicability of their environment to a given task, and help them improve their tools and methods to best suit the environment.

Professor Wartik received the B.S. degree in Computer Science from Pennsylvania State University in 1977, and the M.S. and Ph.D. degrees in Computer Science from the University of California at Santa Barbara in 1979 and 1984, respectively.

INFORMATION SCOPE AND THE OVERSEE CONFIGURATION MANAGEMENT ENVIRONMENT

Steven Wartik
Department of Computer Science
Thornton Hall
University of Virginia
Charlottesville, Virginia 22903

1 INTRODUCTION

This paper studies how *information flow* affects a software development environment. The term "information flow" refers to the transmission of information in an environment. "Information" encompasses all information required to specify, build and maintain software, such as source code, binary code, supporting documentation, forms, or memos; it can be stored and transmitted either electronically or non-electronically. Information "flows" between people, files, and software tools, and between combinations of them—that is, between repositories of information, or between processes that transform information, or both. Information is inherent in software development, so information flow occurs in every software development environment.

The impact of information flow may be defined in terms of information's *scope*. Scope is a measure of how widely information must be disseminated. Some pieces of information have "local" scope; only the people who create them need to know of their existence. Others have "global" scope, meaning that many people must be aware of them to continue with their work. Still others have mutable scope: they are created with local scope, but become increasingly global as the project in which they are used progresses.

Information scope issues equate to deciding where, when and how to transmit a given piece of information during software development. Given the amount of information that goes into developing software, this is a non-trivial problem. It is constrained to some degree by the software process in use, since that process defines the tasks to be undertaken. It is also constrained and aided by the underlying development environment, which usually provides a wide range of tools—electronic mail, project database management systems [11,8], configuration management systems [16,4] that define how information is transmitted, shared and stored. However, these tools are general-purpose. While they can be used to help projects account for scope issues, they neither support nor understand scope *per se*.

The concept of scope provides a simple but useful way to evaluate a software environment's quality. Some environments contain too much information of global scope; these environments overload their users with information, and moreover violate information hiding precepts [10]. Others suffer the opposite problem—very little information of global scope. As has been argued elsewhere [7], this is usually undesirable and slows development. An environment should have scope that supports the software process in use. That is, the software process splits development into a set of tasks,

This research was supported in part by NSF grant DCR-8602674.

each of which requires and produces a given set of information, and requires some set of people and tools. The environment should assign scope that allows exactly those people and tools who need the information to access it, and only at the times that the tasks are being performed.

In this paper, we discuss how environment and tool builders should account for scope. We first define information flow and scope more precisely, and discuss a tool taxonomy implied by scope concepts. Next we cover OVERSEE, a set of configuration management tools for coordinating and controlling information flow and scope, and show how scope concepts and the taxonomy have influenced its design. We conclude with a summary of our experience with OVERSEE, and directions for future research.

2 AN OVERVIEW OF INFORMATION FLOW ISSUES

This section defines information flow more precisely, and discusses its influence in detail. Most of the discussion concerns automated information flow—that is, flow involving a software development environment, using software tools. Non-automated flow, such as oral conversations, is a necessary component of a project, but is harder to regulate. Thus we concentrate on automated flow, but allow for non-automated flow in special situations. In particular, when a high rate of information interchange is necessary, non-automated flow is preferable.

Automated information flow is generally implemented using files, since most operating systems use files as the media for transmitting and storing information. Actually, files are an implementation detail. Our interest is in the information's abstract characteristics; how files implement those characteristics will be ignored. Essentially, we assume that additional attributes and constraints can be attached to a file to give it a data type. Object-based file systems such as those of PCTE [2] and Marvel [5], and the Omega environment's relational approach to storing software [8], implement this concept. [1] enumerates its advantages.

2.1 Information Flow: Definitions and Assumptions

Conceptually, information flow is similar to data flow (*cf.* [3]). It occurs across *paths*, which link a set of sources and destinations. A given information flow path has processes on it that transmit information of a specific data type. That type can be as primitive as bit strings, or as complex as a configuration of compilable source files.

An information flow path may be expressed as a data flow diagram, although it is a special case thereof. In particular, each information flow path has a primary purpose that may be supported by other flow branches. For instance, the purpose of compiling program source is to take source code and convert it to binary code. Doing so, however, also involves the user (for instructions and diagnostics) and program libraries. Thus an information flow path is not necessarily a single path; rather, it is a connected, directed graph, one path through which defines the entire flow path's purpose.

A path may be *physical* or *virtual*. A physical path supports a specific set of activities. It is implemented by some hardware or software constraint within an environment. Like any path, it transmits information of a specific data type, but the type is usually well-defined and meaningful in the context of the activity it supports. For instance, a network connecting two workstations, or the software that manipulates that network, defines a physical path for data transmission. Access to a physical path is often controlled tightly by the hardware of software that implements

it. Networks, for instance, only transmit information in particular formats. As another example, database management systems usually define physical paths that prohibit access to data except through interface tools.

A path that does not exist to support some specific set of activities is termed *virtual*. Virtual paths are generally used for many purposes; as a result, their data types are simple ones (*e.g.*, "string of characters") that permit the flow of most forms of data. Often, an automated virtual path is implemented on top of a physical one. For instance, electronic mail tools define physical paths between users. Information of some "mail message" type is transmitted on those paths. However, they are low-level paths; users take the data contained in a mail messages but interpret that data in the domain of their work, not that of mail protocols. Thus a project using electronic mail for communicating status information uses virtual paths, since the paths could be used for purposes other than those that the project has chosen; the mail tools do not constrain the type of information they transmit. Suppose, however, that the project does not use mail directly, but instead uses tools intended specifically for handling status information, and that these tools are implemented using electronic mail. The tools define physical paths.

Note that an environment contains layers of paths. At the lowest level are physical paths defined by the hardware; above that are paths defined by the operating system; above that are paths defined by tools. As the mail example shows, a path may be physical in the context of one operation, and virtual in another. In general, unless an environment offers a specific tool to perform some operation, that operation must be done using virtual paths.

2.2 Information Flow and Software Development

Any non-trivial project involves a huge amount of information flow. Coordinating it is a serious problem. Environments with too few information flow paths constrain personnel, who cannot transmit information rapidly, or must send it in a roundabout way. Environments with virtual information flow paths provide plentiful opportunities for information exchange, but can do so in an unconstrained way; this often results in a disorderly project. Most systems that attempt to handle information flow relegate it to project management, which is neither fast nor reliable. Many projects experience long delays when management and developers must interact, and this is due to delays in receiving information and responding to it.

An understanding of information flow, then, can improve software environments. An environment with good information flow will contain paths that allow all important information to reach its intended destinations in a timely manner. Little or no consideration of information flow results in an environment where information is exchanged as though the computer were not capable of being used for the purpose. In some configuration management scenarios, for instance, information passes from the development area to the testing area only with the permission of the project's management [12]. The reasons for doing so are fairly clear: management's intervention helps them monitor the project's progress, and helps assure that developers do not accidentally destroy sensitive data. However, the information flow is being halted in the middle of its path, delaying its delivery to its intended destination. Over the lifetime of a project, the total time consumed by such delays can be considerable.

Information flow is also an important component of the software process. Software is usually developed according to a *phased* model—one that separates the development process into activities with distinct goals. A phase requires a predefined set of inputs from previous phases (if any), and

yields a predefined set of products. The nature of the inputs and products is sometimes vague, but it can still be identified. Activities within a phase may be subdivided similarly. Each phase requires a set of tasks; each task requires certain inputs and yields certain products. Some tasks might be refinable into subtasks.

A project accomplishes a task using a specific set of information flow paths. This follows by definition: information exchange must occur in some manner, albeit not necessarily on physical paths. The paths used may or may not be known to the project in advance. The paths might be specific to the task, and used for no other purpose in the environment, or they might be virtual paths used as an expedience; they could also be physical paths used for several tasks. However, information flow paths are a necessary part of project activity.

The paths used in a task can be enumerated. This enumeration is independent of a task's magnitude; it could be something as small as communicating the location of a module (probably requiring a single path), or as large as an entire phase (perhaps needing all paths in an environment). Path enumeration is important when assessing information flow. It assures that information flow analysis can be thorough: because software development may be partitioned into tasks, the paths used for a given subtask, and hence for all tasks that comprise a phase, can be listed. Since information flow arises due to tasks, enumeration also provides a basis for comparing two environments based on their information flow: a comparison is meaningful only in the context of a particular task. Subsequent discussion involving comparisons of information flow paths assumes they are being evaluated with respect to the same task.

3 SCOPE

Information flow may be used to help design the tools in a software development environment. This is done through consideration of *scope*. As the introduction mentioned, each piece of information has an associated scope that defines how widely it must be disseminated. This is measured by how many destinations (people and places) the information must reach; the more destinations, the wider the scope. (A more exact definition is given below.) The concept is important for several reasons. First, it determines how long a given operation takes to complete. The wider the scope of a piece of information (*i.e.*, the more people or places it must reach), the longer an operation involving that information will require.

Second, scope defines the people who can access a piece of information. In a multi-person project, information *must* be shared; that is, information must at some point have scope more global than its creator. An environment should thus assign all information appropriate scope. As we shall see shortly, categories of scope can be defined to aid this process.

Third, scope provides a basis to define the interrelationships between project activities. Each activity produces information that must undergo a scope change to make it available to other activities that require it (presumably, these other activities involve a different set of people). Scope therefore describes coordination of project activities as a function of changes to information. Consider, for instance, two tasks T_1 and T_2, and suppose T_2 requires information I of T_1 (which T_1 is assigned to produce). The traditional model of project activity (see [9], *e.g.*) says that T_2 may begin as soon as T_1 produces I. This is not quite accurate. Rather, I is first produced with scope local to those people assigned to T_1, and the members of T_1 will test I well before they release it. In other words, I exists before it is made available to T_2. Scope supports this concept: the action of releasing I from T_1 is modeled, in scope terminology, by changing I's scope to include the people

assigned to T_2. This may happen while T_1 is ongoing; there is no inherent relation between scope changes and activity termination.

To define scope more precisely, we use information flow paths. That is, the scope of a piece of information is defined by the last information flow path on which it has traveled. An information flow path has one of the following scopes:

1. Local to its originator (the path's primary source).
2. To people within the originator's group (*e.g.*, developers, testers, or project managers).
3. To people in another group.
4. To people in several groups, forming a proper subset of all people in a project, and including the originator's group. (This category may be further subdivided based on the number of groups.)
5. To people in several groups not including anyone in the originator's group.
6. To all people in a project.

This list is in increasing order of scope: we say that a flow path to people within the originator's group has scope that is "more global" than a flow path local to the originator. Increases in scope measure the expected relative time of the information flow path: information of category i may be expected to take longer to reach its destinations than that of category $i - 1$. This follows because people in a given group generally operate within their own sphere of activity, and are not likely to interact with people in another group so frequently as they interact with people in their own.

We may use the above list to assess an environment's consideration of flow. Information must be shared, but the sharing should not be arbitrary. If most paths are global, then the environment is probably attempting too much information sharing and will overload its users with facts they do not need to know. If few paths are global, the environment probably restricts access to information unnecessarily.

Also, a path of greater scope should probably have more access restrictions than one of lesser scope. Paths with scope local to their originator have little or no disruptive potential—other peoples' activities are not influence by them. Conversely, paths with global scope can drastically affect a project. For example, transmitting a module to be tested influences other peoples' schedules, and mistakes in doing so lower productivity. Hence the use of global paths should be controlled very tightly to minimize adverse effects. Similarly, global paths should contain more checks on the correctness of the information that flows on them than local ones.

Certain software process quality issues can be stated in terms of scope. For example, if two activities on a project can possibly be performed in parallel, then the project management might want to require that any information under modification by people involved in the first activity should not be available to those involved in the second. In other words, the information should have scope local to the people of the first activity for as long as that activity is ongoing. This may be enforced in an environment by checking that no information flow paths between people involved in the two activities permit the information to be transmitted.

4 A TOOL TAXONOMY BASED ON INFORMATION FLOW

From the considerations of information flow and scope, we believe an environment's information flow can greatly influence the scheduling aspects of a project. As discussed above, the incorporation of a proper set of physical paths can shorten a project's schedule and reduce mishaps due to miscommunication. Furthermore, we have shown the influence of scope on a project, and proper scope is most easily maintained by the use of physical paths. Since tools define the physical paths, we have created a tool taxonomy that specifies the paths each tool may access. That is, all tools in an environment must fall into one of the following categories:

1. *Information generating tools.* These are used to create and modify information. Text editors are the most common example.
2. *Derivation tools.* These transform existing source information into some other form. Compilers are derivation tools.
3. *Testing tools.* These aid testers in evaluating correctness.
4. *Viewing tools.* These access existing information without permitting modification.
5. *Communication tools.* These provide virtual paths for information exchange.
6. *Information flow tools.* These regulate information flow between project personnel.

Tools in the first three categories all have local scope, whereas tools in the last three have non-local scope. Furthermore, only information flow tools can transmit information that requires people other than the originator to modify their activities. That is, the other tools with non-local scope are used solely to obtain information.

The difference between the first three categories lies in the source and destination of the information required and produced. Information generating tools obtain most of their information from a user, but little or no information from a file system. Information derivation tools, conversely, usually require only the name of a file from a user, then obtain the majority of the information they process from this file. Testing tools usually fall somewhere in between these extremes, since testing requires both a file to test and user-supplied test data. We have found that each category has different information flow path needs. Specifically, the importance of using physical paths is greater in category 2 than in category 1, and greater in category 3 than category 2. This is because physical paths can check the correctness of information more carefully than virtual paths, and the need for correctness in order to complete a task increases from category to category.

The taxonomy is important for several reasons. First, it provides design guidelines for tool builders. Tools with non-local scope are easily identified. Since these tools have more potential effect on a project than tools of local scope, their inclusion should be carefully considered (see below).

Second, it helps in evaluating an environment and a tool's potential within it. Since physical paths are preferable to virtual ones, a tool that cannot be used on a physical path is often undesirable. Moreover, the taxonomy can show the exact set of paths on which a given tool is legal. Such analysis defines the tool's role in a project. It can also aid projects attempting to assess a tool they are considering purchasing.

Third, the taxonomy helps automate project management. An environment with information flow such that communication among project personnel is done largely via physical paths, and with a small set of tools that control access to those paths, can accomplish many management activities that would otherwise need to be performed by humans. That is, much of what constitutes project management is needed only when information is communicated between people. When this communication is automated, the paths can include the verification checks that humans traditionally perform.

Fourth, the taxonomy shows how to increase productivity through efficient information flow. To maximize information throughput, a path's use should be inversely proportional to its scope (since large scope implies increased need for coordination). If tools that access paths of large scope are those used least frequently, and tools that access paths with local scope are used most frequently, people have the most efficient interactions with their development environment. This is true because they obtain information in the most timely way.

5 THE OVERSEE CONFIGURATION MANAGEMENT SYSTEM

To illustrate how the principles discussed in the previous sections apply in practice, we discuss how scope applies in OVERSEE. OVERSEE is a set of software configuration management tools implemented on the Unix operating system [6]. These tools have been greatly influence by scope principles. The result is an environment with more efficient information flow than typically exists in projects with formal configuration management support—information generated in OVERSEE reaches its intended destinations rapidly, and is made available at the times when people are most ready for it.

5.1 An Overview of OVERSEE

OVERSEE tools are information flow tools. They provide a set of physical paths in an environment that control how people interact during a phase. The tools are sufficiently general so as to apply to most software process models, but sufficiently configurable so as to implement the nuances of those models.

OVERSEE assumes three user groups: developers, testers, and configuration managers. Developers create information that is a product of the project. Testers validate that information. Configuration managers are responsible for maintaining the information of both. In this paper, we concentrate on the actions of developers and testers.

OVERSEE does not assume a particular software process model. However, it does presume that software development is phased—*i.e.*, that distinct phases exist. Each phase is an activity that takes some set of products as input, and yields some set of products as output. Furthermore, each phase is presumed to contain some form of testing activity. Since the testing may be quite informal (in a phase to define requirements, it could consist solely of a document review), the assumption is reasonable and adaptable to most software process models.

A phase consists of three activities:

1. Development, where the products are written.

2. Unit-testing, where each product is formally reviewed by its creator.

3. Integration-testing, where the products are merged and reviewed.

While the terminology is taken from programming activities, the activities apply to all phases of development. A paragraph in a requirements document is written and reviewed by a single individual prior to being incorporated in the document and reviewed by others. Note that this view always forces every entity to undergo testing during the phase in which it is created.

A phase is modeled in OVERSEE in terms of a predefined set of transformations. Each entity developed (or modified) during the phase undergoes these transformations. The transformations define who may reference the entity, the tools that manipulate it, and the order in which the references and manipulation occur. The transformations are defined via the OVERSEE tools that implement them.

The current implementation of OVERSEE lets project personnel create any entities they desire in their working area. However, the `configure` tool must be applied to an entity before it is considered an official part of a project. A configured entity is not necessarily a correct, tested entity; if it is program text, for instance, it need not even be compilable. Rather, the tool is used when its invoker first feels an entity is sufficiently defined to be valuable as a part of the system under development. The operation is usually a statement of the entity's existence as a part of the system under development. For example, Ada developers might configure a package specification before it is complete, thereby defining (albeit imperfectly) the general flavor of a package. It is important to realize, however, that the `configure` tool gives its information local scope. Thus only the developer knows of its existence; no one else will be affected by its imperfections.

The entity then undergoes cycles of applications of the `change` and `baseline` tools. These are analogous to the "check-out" and "check-in" operations of version control systems such as RCS [14]: they respectively place the entity in a state where it may be changed, and create a new baseline based on changes made after executing `change`.

Eventually, the entity's developer will have enough confidence in it to want to test it. At this point, he or she executes the `unit-test` tool, and begins unit-testing the entity. Based on these tests. he or she will either `reject` or `accept` the entity. If rejected, it will undergo more repetitions of the (`change`,`baseline`) cycle until it is ready for unit-testing again. If accepted, it is ready for integration testing. The tester assigned to do so will execute the `intr-test` tool, perform tests, and either `reject` or `accept` the entity as part of the product. If rejected, the entity goes back to development, and must be changed, baselined, and unit-tested again. If accepted, it is held in a state where it may be integrated and re-tested with other entities accepted from integration testing. Eventually, when all entities have been integrated and accepted, the product is considered complete, and is given to a configuration manager.

Sometimes, a developer will realize that an entity already accepted from unit-testing contains a flaw. If so, the entity may be recalled to development using the `withdraw` tool.

The `change`, `unit-test` and `intr-test` tools do not actually carry out the operations they name. Rather, they set the stage for the appropriate activity to occur. The actual change to a source entity will be made using an editor or similar tool. Similarly, the testing operations do not necessarily perform any testing; the tools and techniques used to test are project-specific. The OVERSEE tools thus define an entity's stage within the current phase: newly configured, currently undergoing change, baselined, in unit-testing, accepted from unit-testing, in integration testing or accepted from integration testing. A rejected or withdrawn entity reverts to the "baselined" stage.

Figure 1 shows these tools, and their order of invocation, as a state transition diagram. Together, they define the configuration management process as applied to any individual source entity within

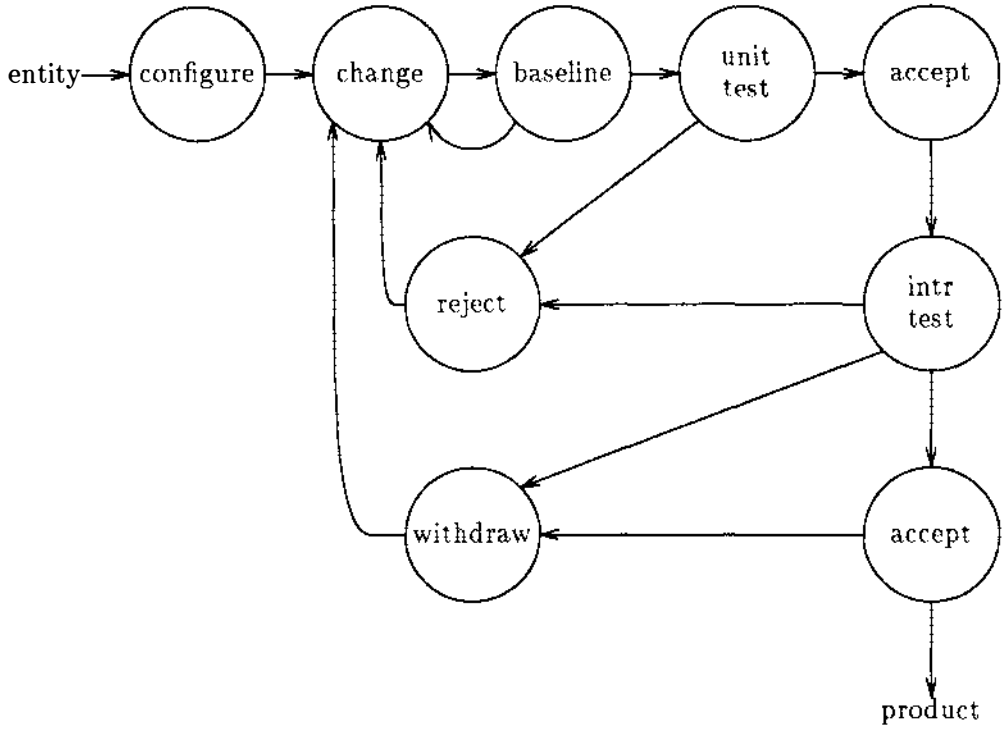


Figure 1: Operation Order in OVERSEE

a system during a single phase of a software process. Because of two freedoms granted to projects—namely, the freedom to define what is necessary for an entity to be configured, and the freedom to define what constitutes testing—they model requirements, design, implementation and maintenance activities equally well.

The operations do not imply a particular software process model. Indeed, we have found the model sufficiently general to handle most major ones [13]. It is most effective when the resultant products of each phase are well defined, because it is aimed toward testing a given entity until that entity is "correct". If the entity's requirements change based on the results of testing (as is common during prototyping), then the scenario is, conceptually, more complicated. The testing process is not as circular as in Figure 1, since the testing process repeats but the same tests cannot necessarily be repeated. Clearly, then, regression testing cannot be used (see below), and new tests must be concocted.

An important feature of OVERSEE is the ability to associate *policies* with each information flow tool. A policy is a precondition for an OVERSEE operation. It may be applied to an entity or a set of entities. It is performed automatically each time an OVERSEE tool is executed; if the argument entity does not satisfy the policy, the OVERSEE tool halts without completing its function. The following are examples of policies we have implemented:

1. A file must contain a version identification number.
2. Each procedure or function must have a comment describing its purpose.
3. Source code must be compilable (no syntax errors).

A policy is associated with a set of OVERSEE tools; it need not apply to all of them. For instance, we associate the first policy listed above with `configure` and `baseline`. Thus an entity must have a version identification when it is first configured and subsequently baselined. Since a user may only change it at those times, this is equivalent to saying that it always has a version identification. We associate the second with only `unit-test`, however. Doing so gives developers maximum flexibility in deciding how many comments to include in initial versions of their code. The third policy, it should be noted, is specific to the implementation and maintenance phases.

We said above that `unit-test` does not perform testing. This is not always true, since policies may be associated with the tool. We have run a project that eliminated most of the code review process by using policies. Not all reviews could be eliminated; a policy can check for the existence of a comment, but cannot verify that it is meaningful. However, an existence check removes much of the drudgery from testing, and catches many simple errors.

5.2 Information Flow in Software Development Environments

The tools in Figure 1 control only the information flow of a single entity, and then only as it relates to tools and to user classes. The figure neglects to show:

1. The information flow paths needed for one source entity to reference others.
2. The paths via which project personnel communicate information about their work.
3. The division of information flow among user classes. For example, an entity will be examined by some subset of the testers; after unit-testing, information flow should therefore occur to the testers, but only to that subset.
4. The information flow between life cycle phases.
5. The information flow as it relates to an entire entity. That is, an individual developer is probably working on only a small portion of a system; a scheme must exist to integrate all entities that make up the system.

We now discuss how OVERSEE handles these issues.

5.2.1 Visibility Issues

We first discuss information flow concepts as they relate to the visibility of an entity—that is, who has permission to access it. This depends on the entity's stage. In OVERSEE, the rights are as follows:

1. An entity not accepted from unit-testing is accessible only by its developer.
2. An entity that has been accepted from unit-testing is accessible to all developers working on the same system and to the testers.
3. An entity that has been accepted from integration testing is accessible to all personnel concerned with the system, and is ready to be installed by the configuration manager.
4. An installed entity is accessible to everyone.

These rules apply to the latest versions of entities (as created by the `baseline` operation), and to entities derived from them. However, different user groups may have different latest versions. For instance, suppose a developer accepts version V of an entity from unit-testing. V then becomes available to testers. The developer, however, might want to add functionality, or respond to reported problems before the testers reject the entity; this entails creating a new version of the entity that is available only to the developer. Moreover, unit-test acceptance is an announcement that a developer considers his entity "reasonably" error-free; for this reason, it should be made accessible to other developers, who might need to integrate it into their part of the system. Thus version V will be available, but successive versions the developer might create will not, until they have passed through unit-testing.

"Availability" is a complex information flow issue in its own right, and warrants some discussion of the flow paths that support it. Developers should be discouraged from making copies of entities they did not create; fewer copies means fewer compatibility problems. Moreover, allowing them source-level access to a entity violates information-hiding principles, which state that they should have access only to an entity's interface, and nothing more [10]. For this reason, OVERSEE encourages information flow not of entities, but of the interface and information derived from an entity. Thus, OVERSEE's information flow tools handle the output of derivation tools. The output of a derivation tool is easily recreated from a given version of a source-level entity, and is usually not easily subject to source-level inspection. Furthermore, it is more easily handled: because the derivation tool may apply certain checks to verify that it is in a reasonable state (for example, does program text compile?), it has formal properties that are not so easily guaranteed of source text.

Strict enforcement of this rule is not possible. In the requirements phase, a source entity is usually uncompilable text, so nothing can be compiled. Thus we adopt the approach that an environment should not encourage creation of copies of the output of information generating tools. The following discussion will show how OVERSEE sets up information flow paths that obviate the need for copies of such entities.

Developers using OVERSEE *never* directly reference the information in other developers' directories. Instead, the operation of accepting something from unit-testing automatically places the necessary information (usually derived information) in an area called the *mini-environment*, which ultimately becomes a library of all information for a project that is past unit-testing. The version of an entity in the mini-environment might not be identical to the latest one in a developer's area, as the above discussion shows. The developer's version, however, is presumed private and, since it is not through unit-testing, unstable. Thus a developer has the flexibility to experiment with and correct her or his own software, while other developers have access to the latest unit-tested version. Under this scenario, a developer can create and unit-test an interface, then proceed to the implementation. Meanwhile, other developers have available the unit-tested interface, allowing them to begin work on their portion of the system.

A remaining issue is what should be available to testers. Testers, like developers, have access to the derived entities in the mini-environment, and thus may test a specific version of an entity. Unlike developers, testers need direct access to source entities; a test for adherence to coding standards cannot be verified by inspection of derived entities. Testers, however, are presumed not to want to make modifications (else they are not acting as testers) and therefore creating a copy of a source entity is not dangerous. Even so, OVERSEE creates the copy "read-only" as a reminder.

However, developers often help other developers. They may also, in the process of integrating entities, discover other developers' bugs. Both situations require direct access to a source entity. Our

solution to this dilemma is to *not* strictly enforce information-hiding through protection. Instead, projects are structured such that a developer has only indirect access to other source entities [15]. This strikes a reasonable balance between environment friendliness, real-world development habits, and information hiding.

5.2.2 Static Information

Information from previous phases is treated separately. It differs from information in the current phase in that it evolves much more slowly, if at all; it is used primarily for reference. Information from previous phases must therefore be available to developers. However, it does not need to be configured such that frequent modification will be needed. This information is therefore termed *static*—it is modified rarely compared to that which is a product of the current phase. Static information is a frequent source of information, but only infrequently a sink. Moreover, static information is modified only with the consensus of many people—developers, managers, and customers. In information flow terms, static information has many outgoing flow paths, but few incoming ones.

In OVERSEE, static entities may be copied but not changed by developers. This facilitates the expansion style of development mentioned above. Changes to an entity must be effected by the configuration manager.

5.2.3 Information Flow During Unit-Testing

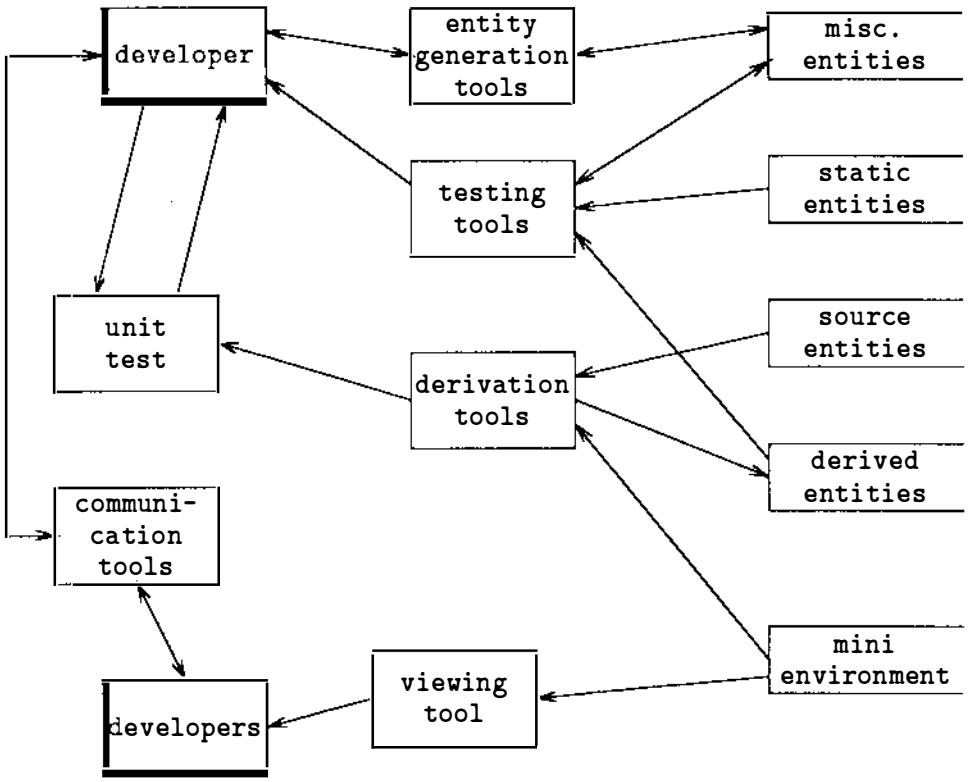
To illustrate the above points, we show in Figure 2 a data flow diagram model of the information flow that OVERSEE uses during the unit-testing stage. The top half is the flow involved in beginning unit-testing (*i.e.*, executing the `unit-test` operation) and performing tests; the bottom half is the flow in accepting or rejecting the entity. The figure is presented in terms of the entity's developer, who performs the testing, the other developers (who may require the entity for their own uses), and the testers assigned to evaluate the entity. It is, in our experience, sufficient to model the automated interactions between them. Note that the sources, sinks, files and processes represent classes rather than distinct items. The testing process, for instance, could be any of several tools, or could be several used simultaneously (the familiar concept of data flow diagram refinement). Moreover, note that one of these diagrams exists conceptually for each developer.

Data flow diagrams do have certain drawbacks that should be noted:

1. *They do not define all the semantics of information flow.* They define the valid flow paths, but not the order in which information flows. This, however, is specified by figure 1, which requires, for instance, that `unit-test` be used before `accept`.

2. *They do not indicate file structure.* Source entities are not "flat." They have a complex interrelationship defined (for programming languages) by module structure. The current OVERSEE implementation is on Unix, and helps developers use Unix's hierarchical file system to indicate unit boundaries. Data flow diagrams reveal neither of these facts.

Despite these problems, data flow diagrams have an appealing simplicity that makes them well-suited to the task at hand. The necessary information can either be inferred or defined using another model.



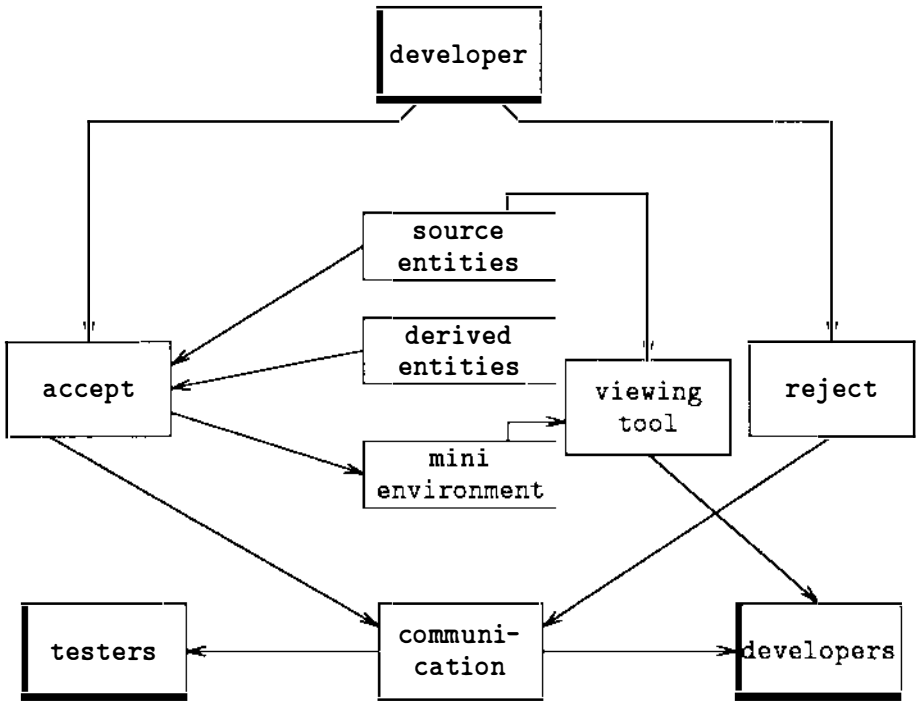


Figure 2: Unit Testing: Information Flow

Figure 2 groups entities into four categories: derived, source, static, and miscellaneous. The first three have already been defined. The fourth—miscellaneous entities—represents any nonconfigured entities that might be present. Recall that an entity is not considered "source" until it is configured; therefore, all source entities begin as miscellaneous. Also, test cases that a developer creates are not required to be configured (although policies can enforce this); therefore, test case files are miscellaneous entities.

The diagram does not specify how static information and policies are created. Since static entities should be available for reference, beginning a phase should place them in a globally accessible place. In OVERSEE, this is automatic. Policies are similar: in a well-run project, policies are entities in their own right. Hence they undergo their own development and testing phase. In the phases where they apply as policies, they become a special type of static entity. Therefore they are automatically placed in position at the beginning of each phase.

6 SCOPE ISSUES IN OVERSEE

The OVERSEE tools have more functionality than was previously indicated. Much of this functionality supports information transmission between users: OVERSEE automatically notifies appropriate parties when operations occur that affect them. In particular:

1. When a developer executes `unit-test` on an entity, those people assigned to integrate and test the entity are notified that it is almost ready.
2. When a developer executes `accept`on an entity, those people assigned to integrate and test the entity are notified that it is ready.
3. When a developer executes `reject` or `withdraw`on an entity, those people assigned to integrate and test it are notified of the problem.
4. When a tester executes `intr-test`, the developers of the entities being tested are notified, as are people involved in subsequent integration and testing steps.
5. When a tester executes `reject`, the developers of the rejected entities are informed, with an appropriate statement as to why the entity was rejected.
6. When a tester executes `accept`, those involved in the next step of integration and testing are notified.

Automatically notifying people of an operation's outcome is not unique to OVERSEE. Marvel [5], for instance, incorporates this ability. However, Marvel only provides notification mechanisms; our aim is to study what is most appropriate for configuration management operations. This problem can be phrased in terms of scope, by discussing when information must change scope, and why.

OVERSEE tools are information flow tools. They provide physical information flow paths between developers, and assure that those paths are used only at the correct times. Not all OVERSEE tools have non-local scope, however. That is, the ones with local scope do not define physical paths to other users. However, they are still important as information flow tools, because they set the stage for subsequent OVERSEE operations.

The `configure`, `change` and `baseline` tools have local scope, meaning they access no information flow paths of non-local scope. The other tools have non-local scope. `unit-test`, `accept` and

`reject` (developer's version) access paths to testers (hence, people in another group). `intr-test`, `accept` and `reject` (tester's version) access paths to both testers and developers; hence, a subset of people in several groups, including the originator's group.

These scopes reflect decreasing order of command use. That is, commands in the first group are executed more frequently than in the second, and in the second more frequently than the third. This occurs because developers usually find most problems early on; therefore, fewer bugs are found and corrected per execution of `intr-test` than per execution of `baseline`. This is another reason for `baseline` to have local scope. Developers do not execute non-local scope commands until they have some confidence in their software. This saves on unnecessary information flow: we have found that they catch and remove many careless errors before accepting their software from unit testing. That is, they remove the errors while the information still has local scope. If the errors were instead caught by someone else (when the information has non-local scope) then the error report must involve at least two people, and make take some time due to the need to coordinate their interaction. Removing errors when information has narrow scope is highly desirable.

Policies play an important role in this process. We associate only a few policies with the local-scope tools, and thereby give developers freedom in how they build their software. However, the more global a tool's scope, the more policies we attempt to associate with it. Thus, to enter unit-testing, an entity must meet many previously unimposed constraints. Because integration-testing involves more people than unit-testing (and thus potentially longer time delays), even more policies are imposed when integration-testing begins.

Scope is also related to the current stage of an entity, and the permissions one has with respect to it, as defined in figure 1. Initially, an entity's scope is local to its developer. In time, the scope is extended to other developers on the same project who might find it useful. Subsequently, the scope encompasses testers, who are asked to evaluate it. Finally, the entity is available to everyone.

Note that an entity with global scope may be examined by tools with local scope. A tester, for instance, might use a static analyzer on some entity. By definition, that entity must have non-local scope. However, executing a static analyzer does not directly affect anyone else; in that context, then, the entity is being treated as having scope local to the tester. Any tool without side effects—that is, a testing, viewing, or communication tool—may be used freely on an entity, no matter what its scope.

In general, user-activated OVERSEE commands are executed infrequently in relation to the total number of commands a developer invokes. The basic activities of generating source entities, preparing test cases, *etc.*, consume far more time than do configuration management activities. However, each OVERSEE command tends to represent a significant step in the software process. Therefore OVERSEE commands usually perform many activities that are not visible to the user unless they fail. These activities include policy checks, inter-module consistency analysis, and notification of changes to other users.

OVERSEE tools do not deal with every possible information flow path in an environment. Fortunately, they need not be comprehensive. Environments already contain suitable tools—file viewing and copy utilities, mail services, compilers, *etc.*—for dealing with many information flow paths, especially those that involve information review rather than modification. Judicious use of directory structures and protection mechanisms can define many of the information flow paths. OVERSEE provides the remainder.

7 EXPERIENCE AND CONCLUSIONS

Many of the problems in software configuration management stem from improper communication channels between project personnel. The lack of good information flow paths causes lost information, schedule delays, and misunderstandings about project components. However, it is hard to assess what makes a path "good." We have presented information flow scope as a means to judge just how good the paths in one's environment are.

The OVERSEE environment, the design of which was greatly influenced by considerations of scope, has been successfully used to manage software configurations of several projects. The largest one to date involved the design and implementation of a small accounting system. The project, consisting of eighteen people, had the responsibility to produce (and validate) requirements, design and code within a 12 week period. Ten acted as developers, and ten as testers. The two groups were organized as egoless teams; the usual warnings of a six-person maximum group size were deliberately ignored to induce large amount of communication.

Many scope levels existed in the project. Each individual dealt with local scope, scope involving her or his subgroup, the entire group, and the whole project. Furthermore, the organization of the project shifted due to personnel constraints, introducing further complexities.

OVERSEE was used in all stages of the project, and in particular handled all communication involving project status. Other tools that the students were allowed to use (compilers, editors, debuggers) were carefully classified and placed on appropriate flow paths. The best results were achieved during the coding phase, when we were able to run several rounds of unit and integration testing on each module within the system, all within the space of a week. Since the project was undertaken in a university setting, where people have no common office space and thus have less opportunity to communicate than is typical (in fact, one person was located one hundred miles from the site), organized information dissemination was crucial. People used electronic mail at times, but only to clarify issues on functionality. This seems good proof that the taxonomy is realistic, in that a few tools can suffice to handle the inter-person information flow.

References

[1] Robert Balzer, "Living in the Next-Generation Operating System", *IEEE Software*, 4(6):77–85, November 1987.

[2] Ferdinando Gallo, Regis Minot, and Ian Thomas, "The Object Management System of PCTE as a Software Engineering Database Management System", *SIGPLAN Notices*, 22(1):12–15, January 1987.

[3] C. Gane and T. Sarson, *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall, Englewood Cliffs, NJ, 1979.

[4] Patrick Hawley, "DACOM: A Design and Configuration Management System", in *COMPSAC'83*, pages 580–587, Chicago, IL, November 1983.

[5] Gail Kaiser, Peter Feiler, and Steven Popovich, "Intelligent Assistance for Software Development and Maintenance", *IEEE Software*, 5(3):40–49, May 1988.

[6] Brian Kernighan and John Mashey, "The Unix Programming Environment", *IEEE Computer*, 14(4):12–24, April 1981.

[7] Mark Linton, "Distributed Management of a Software Database", *IEEE Software*, 4(6):70–76, November 1987.

[8] Mark Linton, "Implementing Relational Views of Programs", *SIGPLAN Notices*, 19(5):14–21, May 1984.

[9] Harlan Mills, Richard Linger, and Alan Hevner, *Principles of Information System Analysis and Design*, Academic Press, Orlando, FL, 1986.

[10] David Parnas, "A Technique for Software Module Specification with Examples", *Communications of the ACM*, 15(5):330–336, May 1972.

[11] Maria Penedo, "Prototyping a Project Master Data Base for Software Engineering Environments", *SIGPLAN Notices*, 22(1):1–11, January 1987.

[12] Maria Penedo and Arthur Pyster, "Software Engineering Standards for TRW's Software Productivity Project", in *Proceedings of the 2nd Software Engineering Standards Application Workshop*, San Francisco, CA, May 1983.

[13] Helen Roh, *A Study of Information Flow in Software Development Environments*, Master's thesis, University of Virginia, Charlottesville, VA, 1988.

[14] Walter Tichy.. "Design, Implementation, and Evaluation of a Revision Control System", in *Proceedings of the 6th International Conference on Software Engineering*, Tokyo, Japan, September 1982.

[15] Steven Wartik, "File System Structures for Object-Oriented Development", in *Proceedings of 5th National Conference on Ada Technology*, pages 411–419, Washington, D.C., March 1987.

[16] Sandra Zucker, "Automating the Configuration Management Process", in *SOFTFAIR*, pages 164–172, Arlington, VA, June 1983.

SOFTWARE CONSTRUCTION PROJECT MANAGEMENT:

MATCHING MANAGEMENT AND METHOD FOR SUCCESSFUL SOFTWARE PROJECTS

Robert E. Shelton
CASEware, Inc., of Oregon
13050 S.W. Forest Glenn Court
Beaverton, Oregon 97005

ABSTRACT:

Project management and software engineering methodology are basic tools in the quest to deliver correctly working software on budget, on time, every time. Pairing proven engineering methods with proven construction industry project management techniques is offered as an alternative to traditional, less structured approaches to software development.

BIOGRAPHICAL SKETCH:

Robert Shelton is Vice President and co-founder of CASEware, Inc., a leading provider of management advisory services and business solutions. CASEware specializes in rapid development of high quality software using Computer Aided Software Engineering (CASE) tools and the Data Structured Systems Development Methodology (DSSD).

Shelton's background is both technical and managerial, including five years as a project manager with Pacific Bell; and two years engineering systems for the Stanford University Graduate School of Business. Over the past ten years, Shelton has operated two private businesses, and actively participated in the formation and operation of two corporations.

Shelton received his Bachelor of Arts degree in Psychology from Stanford University, specializing in neuropsychology and computer science. He is a member of Institute of Electrical and Electronics Engineers, Project Management Institute and the National Forensics League.

INTRODUCTION

Information Systems is the albatross of American management. Each year $40.3 billion is expended developing new business software; another $33.6 billion is consumed by maintenance and repair -- much of that due to failure of the initial development project to meet objectives (1). The General Accounting Office determined that 15% of the information systems projects reviewed in one recent study were paid for in full, but never delivered any product at all. Fewer than 2% of the projects studied delivered the specified product in working order (2).

To put this cost in perspective, consider that each year, we spend as much on information systems software repair as NASA spent in an entire decade to put Americans on the moon.

With fewer than 5% of software projects using project management and only 15% using a software development methodology, such statistics are hardly surprising (1).

The author offers an alternative.

Application of proven construction industry project management techniques combined with a proven software engineering methodology produces systems built to cost, schedule and specification. We refer to this approach as Software Construction Project Management.

METHOD TO OUR MANAGEMENT

When project management methods are applied to only 5% of today's software projects, software development is not being project managed.

Caveat: Successful projects do not just happen; they are project managed.

Project management is a methodology. Webster's defines a methodology as a body of procedure or process for attaining an object; methods, rules and postulates employed by a discipline (3). A methodology is a set of practices which guide one to meeting one's objectives.

The methodology of Project Management must manage three fundamental factors within project priorities, constraints and available resources:

COST: The tension among project budget, benefit (savings, profit) realized from completing the project as specified, and the opportunity cost of not undertaking the project.

SCHEDULE: Time available before project value (savings, profit) is exceeded by project cost vs the time required to deliver the performance required to deliver that value.

PERFORMANCE: The results required to achieve the value desired, based on specific, identified assumptions.

Unfortunately, what passes for "project management" on most software projects is nothing more than task scheduling. (Therein lies another problem: without a software engineering methodology, how can one define the necessary tasks to schedule?)

Construction industry project management, on the other hand, is a set of practices which guide project managers in delivering their products working as contracted within agreed upon budget and schedule constraints. In the construction industry, project management is so ingrained it is taken almost for granted.

Consider the results achieved as evidence of the effectiveness of construction industry project management. Compare the notable absence of abandoned, partly completed skyscrapers to the frequency of abandoned, partly completed data processing projects. While the construction industry has experienced some notable failures, no parallel exists to the chronic project budget and schedule overruns of 100% and 200% in software development. The very low rate of failure in construction suggests the industry has developed project management practices which work.

Central to its success, construction project management constrains builders to performing the necessary tasks in the correct order. By analogy, try building the Bonneville Dam by pouring concrete without blueprints! What construction firm would even consider pouring concrete without a design specification? In software development, this is the norm. Software concrete is called "code". Lacking both project management and software engineering methodologies, most software projects start at the wrong place: pouring concrete. Industry jargon for this practice is "design on the fly". "Protocycling" is the more fashionable name.

Since the construction industry has about the same percentage of project failures that software development has successes, the software industry could profit from modeling itself after construction practices. The task at hand, then, is to apply the construction industry's successful project management techniques to software development projects. To accomplish this task, we should first establish the basis for transfer of project management technology.

PROJECTS HAVE A LIFE (CYCLE) OF THEIR OWN

Projects in all industries involve the same steps or stages, thereby allowing technology transfer from project management successes in other industries to software development projects. The author makes these assumptions:

1. A project is a project, regardless of the product or industry.

2. Successful project management is successful project management, regardless of the project.

Thus, software development is a project, the product of which is -- or should be -- working software. With a proven engineering methodology, a software project can be managed as successfully as a construction project. The methodology imperative in conjunction with project management must be made explicit in the software business. Construction industry practitioners follow a method so inherent as to be inseparable from the industry itself; method in software is not de rigueur.

Comparison of proven models for project management and software engineering substantiates these assumptions. This comparison becomes the bridge from project management to engineering method that is taken for granted in the construction industry. This bridge is the basis for implementing the management-method approach of Software Construction Project Management.

The LaFleur project management methodology (4) specifies a project life-cycle and the tools required to manage a project within the project model, regardless of industry or product. This approach is based on the application of project management in such diverse industries as construction, automotive manufacturing, electronics engineering and shipbuilding. Because LaFleur's model is well defined and broadly applied, it is used as a representative model for the methodology of construction project management.

LaFleur's TSRPI project model concludes that any and every project experiences a common life cycle -- even if faulty management attempts to "skip" a step. (See Diagram #1)

Ken Orr promotes a systems development model, Data Structured Systems Development (DSSD), which identifies what must be done and in what order to build information systems (5) -- to wit, the project life-cycle model for software. (See Diagram #2)

DSSD is a methodology for engineering systems. Approaching software construction as a science instead of a creative art achieves working systems because the systems builder has a set of manageable tasks (i.e. a project) instead of an undefinable, amorphous invention.

The parallel between DSSD and TSRPI as approaches to managing a project through the life-cycle suggests the difference is in the product, not the process. (See Diagram #3)

This parallel links management to method, while also demonstrating the viability of construction project management in a software engineering environment. The steps in the systems methodology are the tasks which must be completed in a specific order to take a problem definition to a completed and working software product -- the tasks of the project that are managed. Construction project management picks up from there. Given a defined set of tasks, project management can be applied to manage the assumptions which underlie the tasks. This is basic project management as practiced outside the software industry.

TO HACK OR NOT TO HACK, THAT IS THE QUESTION!

In the construction industry, the architectural process (analysis & design of buildings) is an art refined to a science. While the results express tremendous creativity, the process is so well defined that debates rage over whether or not the aesthetic style of a particular building "violates" the principals of "good architectural practice". Once a building design is complete, the implementation process is rigorous execution of the blueprints.

While such practices are not new in the construction industry, fundamental attitude changes will be required among software developers and managers to bring software development up to the status of software "engineering" (6,7,8,9,10,11). The value of software engineering methods and sound project management practices seem to be viewed in the industry as:

1. OBVIOUS, but we don't have time to do it because we're coding right now, or

2. IRRELEVANT because software results from creativity that's beyond management and engineering (6,10).

Fully 95% of all software projects are evidently attempting to live out the illusion that software is art, ignoring the opportunity for engineering and project management.

Software engineering methodologies are an alternative to the "invention and creativity" approach. A software engineering methodology is a systematic approach to taking a business problem or need and producing a workable systems solution -- not simply the life-cycle stages, but the practices, rules and standards for modeling, blueprinting, and building. The objective of a software engineering methodology is to achieve construction-industry like levels of process reliability, from rule constrained design to precise implementation of the plan into code.

A sound methodology provides tools to represent and communicate the process and results at each step. Communication is critical to a successful project management. A sound methodology also makes the steps in the project tangible and measurable, so progress can be measured against schedule, budget and expected performance.

While DSSD is our preferred example of an effective methodology, it is just one proven approach among several to engineering systems. The critical factor is not WHAT methodology is used, but that a proven methodology IS rigorously used.

SOFTWARE CONSTRUCTION

The logic and necessity of combining project management and a software engineering methodology becomes evident through comparison of the construction project life-cycle to that of software engineering.

Step #1: CONCEPT

Conceptualization of a building or a software system is the "great idea" -- the wish or need that "just" needs to be implemented.

Concept is the desire to build a new corporate headquarters, such as the Seafirst Columbia Center; or the strategy to computerize particular business operations to reduce labor requirements. Concept is referred to as Systems Planning in DSSD because the ideas for possible software projects, their interrelationships and impact on the business are developed at this stage. Such processes are comparable to developing facilities strategy, alternatives and ideas for housing the corporation.

Step #2: SCOPE

Scope is the refinement of Concept -- the honed idea.

In construction of a corporate headquarters, certain constraints of size, budget, timing, and environment are critical. Similar issues are relevant to Scope in software engineering -- so called Project Planning.

In DSSD, Scope is also preliminary business context, the participants and transactions which make up the business operations in question. While context is itself intangible, it can be represented with an Entity Diagram. From business context, function can be determined. Function is the sequence of business processes required to obtain the end result.

Context and function in DSSD are the systems equivalent of the preliminary decisions about form and function for a building. They are the what and how at a planning level.

Step #3: MODEL

Three types of models may be used to precisely define a building or a software system. The industries implement models somewhat differently, but the concepts don't differ:

PAPER MODEL	the "artist's conception", frequently sketches, drawings; "concept pieces";
NON-WORKING MODEL	the static "scale" model that looks like the building or system being built, but has no functionality what so ever;
WORKING MODEL	the semi-operational prototype; detailed scale model with critical parts working;

Modeling, in its various forms, is the heart of software requirements definition -- developing a picture of the need.

Step #4: BLUEPRINT

In construction, blueprints are exact specifications of what to do, where, how, and what materials to use in the process. Blueprints are precise specifications of all details. Software blueprints are surprisingly similar: what on-line screens and printed reports must do and look like; how each process must operate, connect to others, relate to the data base; structure of the data base, element attributes, relationships and edits.

Step #5: CONSTRUCT

Construction is the most familiar and obvious part of building an office tower -- and the easiest part of systems building to "see" as well. As construction is also the only visible "product" of software development, software managers become remarkably uncomfortable when "Johnny's not coding."

Obvious from this sequence of steps, however, is the real issue in construction: the preparation. The purpose of planning is the value derived -- what can be done, built, understood because of the concept, scoping, modeling and blueprinting. Construction is doing what has been planned. This is precisely what is missing in software development today.

WHAT HAS BEEN LEARNED FROM APPLICATION OF THESE CONCEPTS?

Several pitfalls have been identified from project managing software engineering. These pitfalls exist whether or not construction project management or a software engineering methodology is applied to a particular software project, and can disrupt even a well managed project. The project manager is, however, in a better position to identify and manage around these pitfalls when applying the techniques discussed in this paper.

Pitfall #1: Software is constructed of invisible bricks.

The product of building construction is physical. The status of a construction project can be assessed by counting the bricks. When half the required number of bricks have been laid for a wall, the wall is half complete.

What in software can be counted? The product of an information systems project is not physical -- it is positively not lines of code or program modules. The product of software development is transformation of information. If the right data goes in, but the wrong answer is delivered, the transform is wrong -- not "95% complete". Defective transforms simply do not work. Period. We have no tool to measure how close the transform is to completion until it works and is thus 100% complete.

Having a non-tangible product complicates assessing schedule status, product quality, resource sufficiency, and plan validity. DeMarco asserts "You can't control what you can't measure" (2). Yet the tools for measuring the software product are primitive and little used. Somehow, the operational principal has been "You can't measure what you can't see."

Logical modeling offers the most leverage with the intangible. If we cannot see the product itself, we can model it with representations that can be seen. The challenge is building CASE tools which will directly, reliable, and completely convert the model to the transform -- so-called systems generators.

Pitfall #2: Software models masquerade as "The Real Thing".

The value of software modeling is the simplicity, low cost, short build time, and throw-away nature of the working model. In software we can build a convincing model of a very large project in a hours or days. For this same reason modeling is used extensively in construction. A "good" working model may look like and even work like the intended final software product.

Successful models create a dilemma: If the prototype looks like it works as the real system should, clients and managers rarely can understand why significant work is required to deliver a production quality product. The model may be nothing more than a series off artificially manipulated pictures which fake the intended features of the production system, but it looks "real". While in construction, no one would confuse a desk-top model of the Empire State Building with the actual structure, the same is not true with software.

Working software models put the developers in the dilemma of having to justify why they insist on following a proper methodology instead of leaving the users with a prototype that looks real, and therefore, must be "almost finished".

Pitfall #3: Software lacks 2x4s.

In construction there are standardized components, of which the 2x4 is an excellent example. In software we lack standard, off the shelf components from which to construct our products (12).

Custom fabrication is more costly than off the shelf components because the amortization unit base is one. The dependence of software development on custom fabrication explains both the very high cost of software manufacture and the relatively low quality of the finished product. The software industry must reinvent because we are not yet equipped to reuse.

Pitfall #4: Software development lacks industry standards.

Most industries have a body of accepted "industry-wide" standards. These standards specify what types of materials to use for different purposes, what degrees of reliability and quality to expect, and even what deviations from the standards are acceptable. Building, electrical and plumbing codes are examples from construction. There are no industry-wide materials or quality standards for software.

One of the characteristics which differentiates engineering from art is standards. The widespread use of any good systems development methodology is one of the most fundamental standards which stands between artistic and engineered software.

A lack of industry-wide standards, however, may only belie a more fundamental problem. Industry-wide standards are a non-essential abet ideal goal. The development of company- or organization-wide would suffice. The construction parallel is local building codes, which vary slightly from city to city depending on local requirements.

Pitfall #5: Software estimation is not an engineering practice.

Estimating the budget and schedule for software projects is not one of the software industry's strengths. The situation is complicated by the inability to see what we are measuring, as well as industry practices. Research by DeMarco shows an industry-wide ten-fold variance in programmer productivity (2). This results from dependence on custom fabrication. As with any craft based on hand-construction of a product, some practitioners will be wizards, and others not so wondrous.

A good software estimator's handbook to support reliable budget and schedule estimation awaits industry standard software engineering practices. Only with standards and methodology will performance become predictable. At such point, the software equivalent of a Blue Book can be written.

WHAT ARE THE IMPLICATIONS OF THIS APPROACH?

Experience applying Software Construction Project Management building systems for industry and government alike suggests several potentially significant implications:

1. Software can be engineered to specifications because the problem to be solved can be defined clearly enough to specify a correct solution.

2. Software budgets and schedules can be estimated within tolerances of +/- 10% because the steps to project completion can be precisely and completely defined. Estimation quality can be high enough to use as the basis of legally-binding contracts, as is the practice of European software developers (13).

3. Existing, proven project management techniques provide the leverage necessary to implement the steps in a software engineering methodology.

4. The combination of construction industry project management methods and a proven software engineering methodology produces systems build to cost, schedule and specification.

5. The critical issue in matching management and method is not WHAT methods are used, but that proven methods ARE rigorously applied to every software project.

But implementation of this approach will require changing the behavior of the software profession. To apply construction project management methods and software engineering methods, software building must be viewed as a process of engineering, not art.

DIAGRAM #1: LaFleur TSRPI Life Cycle Model

The LaFleur model presents five project life cycle phases: Think, Study, Research, Plan and Implement (TSRPI).

Although different names may be used, or a given organization may elect to name their projects by another name, LaFleur reasons that all completed projects "live" each phase.

Each phase represents a new level of refinement on the previous, drawing in new issues until the project is implemented.

LaFleur does not consider the last phase to be part of the project, per se. In this model, Operations results from and is supported by Projects.

PHASE	CONCEPT
Think	Synthesis of Objectives (projects) from Goals and Ideas
Study	Evaluate objective for strategic fit in business goals
Research	In-depth examination of idea, resources, buyers/users needs
Plan	Identify in detail what must be done & how to do it
Implement	Execute the Plan
Operate	Use project result to achieve Objectives in support of Goals

DIAGRAM #2: Orr DSSD Life Cycle Model

The Orr Data Structured Systems Development (DSSD) model represents project phases in systems terms.

Like TSRPI, DSSD is an iterative or waterfall model of the project life cycle. Each phase refines and builds upon the previous until the project is implemented.

Orr reasons that even projects which start "coding" immediately must proceed through these steps. The reason, Orr concludes, that such "coding" projects frequently fail is that the code itself becomes a constraint on the process by virtue of its existence.

In more traditional models, the distinction between software maintenance and Operations is often blurred. Operations, in the Orr model is actual use of the software product. Orr views software maintenance either as repair of incorrect implementation (the product did not work correctly or no clear specification was developed in the first place), or as new development (a new report is required due to business changes).

PHASE	CONCEPT
Systems Planning	Create Objectives (projects) which address business Goals
Project Planning	Specify goals, objectives, business scope, buyers & users
Requirements Definition	Evaluate business needs, functionality & alternatives
Analysis and Design	Detail how solution will be built and how it will work
Build/Test/Implement	Construct the software based on Design
Operate	Use of software by the buyers/users who commissioned the project

DIAGRAM #3: Comparison of TSRPI and DSSD Life Cycle Models

Comparison of the LaFleur and Orr models around a backbone of more generic concepts manifests a strong symmetry.

This symmetry is the basis for matching management and method in Software Construction Project Management.

LAFLEUR MODEL	CONCEPT	ORR MODEL
Think	Idea Concept	Systems Planning
Study	Rough CSP Scope Domain	Project Planning
Research	Model Description Constraints	RequirementsDefinition
Plan	Blueprint ConstructionPlan Detail CSP	Analysis & Design
Implement	Construct & Install	Build, Test
Operate	Use the Result	Operate

BIBLIOGRAPHY

[1, Jones, 1986]:

Jones, Capers;
Programming Productivity - Issues for the Eighties,
Second Ed.;
IEEE Computer Society, Piscataway, New Jersey, 1986.
(Data used from publications and personal communications)

[2, DeMarco, 1982]:

DeMarco, Tom;
Controlling Software Projects - Management, Measurement
and Estimation;
Yourdon Press, New York, New York, 1982.

[3, Webster, 1983]:

Webster's Ninth New Collegiate Dictionary;
Meriam-Webster Inc., Springfield, Massachusetts, 1983.

[4, LaFleur]:

LaFleur, Ronald I.;
Project Management with Ronald I. LaFleur;
Project Management Assistance Corp., Scituate, Masss.
(Concepts used from seminar materials)

[5, Orr, 1986]:

Orr, Kenneth;
Data Structured Systems Development Design Library;
Ken Orr & Associates, Topeka, Kansas, 1986.

[6, Brooks, 1975]:

Brooks Jr., Frederic P.;
The Mythical Man Month - Essays on Software Engineering;
Addison-Wesley, Reading, Massachusetts, 1975.

[7, Beck-Perkins, 1975]:

Beck, Leland L., and Perkins, Thomas E.;
"A Survey of Software Engineering Practice, Tools,
Methods, and Results",
IEEE Transactions on Software Engineering;
IEEE Computer Society Press, Piscataway, New Jersey, 1983.

[8, Jones, 1984]:

Jones, Capers;
"Reusability in Programming - A Survey of the State of the Art",
IEEE Transactions on Software Engineering;
IEEE Computer Society, Piscataway, New Jersey, 1984.

[9, McNamara, 1986]:

McNamara, Don;
"Japanese Software Factories", Proceedings of FEEDBACK86;
Ken Orr & Associates, Topeka, Kansas, 1986.

[10, Metzger, 1981]:

Metzger, Philip W.;
Managing a Programming Project, Second Ed.;
Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[11, Silverman, 1985]:

Silverman, Barry G.;
"Software Cost and Productivity Improvements - An Analogical View", COMPUTER;
IEEE, Piscataway, New Jersey, May 1985.

[12, Ledbetter-Cox, 1985]:

Ledbetter, Lamar, and Cox, Brad;
"Software-ICs", BYTE;
McGraw-Hill, Inc., New York, New York, June 1985.

[13, Dawson, 1987]:
Dawson, M.;
"Iteration in the Software Process",
Proceedings of the 3rd International Software Process Workshop;
IEEE Computer Society Press, Piscataway, New Jersey, 1987.

Structured Management of Software Development Projects

William H. Roetzheim

Abstract

This paper describes a structured approach to computer project management which is significantly more rigid than current approaches. The techniques presented match those described in the author's book *Structured Computer Project Management* (Prentice-Hall, 1988).

Author's Biography

Mr. Roetzheim is the author of several books, including *Structured Design Using HIPO-II* (Prentice-Hall, anticipated late 1988), *Structured Computer Project Management* (Prentice-Hall, 1988), and *Proposal Writing for the Data Processing Consultant* (Prentice-Hall, 1986). He has worked as Senior Project Manager for a Honeywell subsidiary and is currently employed as a member of the technical staff for The MITRE Corporation, a Federal Contract Research Center. Mr Roetzheim has a Masters degree in Business Administration (MBA) and is a Masters candidate in Computer Science.

The author's mailing address is:

William H. Roetzheim & Associates
3891 American Avenue
La Mesa, CA 92041
(619) 464-0182

Structured Management of Software Development Projects

Introduction

Software project management is often considered an *art* which must be learned from experience. The author feels that software project management can be taught in a step-by-step, logical fashion. The resulting *science* of software project management is termed structured computer project management.

I would estimate that nine out of ten project managers I've met use casual project management. Every project is managed a little differently. They seldom know what aspect of the project they will be working on later that day, much less in six months. They are familiar with many of the tools available to them but use them inconsistently. Projects often degenerate to the point where crisis management is the rule rather than the exception. Effective delegation of project management tasks is truly impossible because the tasks are not sufficiently well defined. The only way to train new project managers is a baptism by fire, starting with small projects and graduating to larger projects as experience is gained.

This is not to say that the project managers I'm referring to are not successful. Quite the contrary, many of them are leaders in their field using just this approach. Their experience gives them the intuition needed to get the job done in this fluid environment. In spite of this, I contend that casual project management should be banished from the industry! Although the casual approach takes less work on your part, I believe that formal techniques are worth the effort. Let me present four reasons why I am so strongly opposed to an apparently successful approach to project management, and why I wrote the book *Structured Computer Project Management* (Prentice-Hall, 1988).

Reason 1: The increasing size and complexity of computer software development projects is rapidly making effective casual project management impossible.

Reason 2: Consistent, predictable performance is not possible using casual project management.

Reason 3: Using casual project management, it is not possible to effectively delegate project management tasks while still retaining adequate control over the project.

Reason 4: The learning curve for new project managers is unacceptably long when using casual project management.

Significant Phases of a Computer Project

Some of the worst data processing project managers around are competent during all phases of software development. How is this possible? Project management includes tasks which are not considered part of software development. The software development life cycle typically consists of design; code; test and deliver; and maintenance. The project management life cycle consists of analysis and evaluation; marketing; design; code, test and deliver; and postcompletion analysis. Project analysis and evaluation is conducted

prior to contract award, and involves analyzing the project requirements to determine if it is technically accomplishable, if the risks to your company are justified by the benefits of doing the work, and if the customer's schedule and cost expectations are realistic. On most mid-sized and larger projects, the analysis should also include financial projections for the life of the work. Financial projections normally include predicted monthly cash flow, income, expenses, return on investment, return on assets, and asset turnover. The modeling necessary to perform this analysis is fully described in [Roetzheim, 1988a].

Project marketing involves assisting with the preparation of a technical proposal and reviewing costing/schedule commitments. Marketing computer projects is described in depth in [Roetzheim, 1986].

Clearly, the project manager becomes involved well before software development commences, diverging from the software life cycle after system delivery. A project manager who ignores project analysis and evaluation, project marketing, and postcompletion analysis is seldom successful.

Effective Project Planning

Have you ever been unpleasantly surprised when a critical milestone date for your project arrived and the required product was "not quite done". Have you ever been late with a software development project and experienced that sinking feeling as you realized that you could not give the customer an honest answer about how late you would be? Have you ever managed a project that used 90% of the time to do 90% of the work and another 90% of the time to do the other 90% of the work?

The cornerstone of project tracking and control is a carefully thought-out project plan. The *project plan* is a formal or informal document which is both a tool used by you throughout the project and a deliverable item submitted to your management. It is also common for the project plan to be submitted to the customer, and for relevant portions of the project plan to be available to the project team. The project plan normally includes:

- Definitions of project activities and required results (deliverables)
- Dependency definitions for tasks
- Resource estimates for each project activity, often expressed in man-days for people
- Risk estimates for each activity, and plans to reduce the risk or the impact of failure for high-risk tasks
- Activity schedules which include and highlight milestones and other checkpoints for formal progress reviews
- Resource budgets and loading estimates (resource requirements by time period) for personnel, equipment, and other resources which must be tracked

A high percentage of the software development work involves the requirements analysis and preparation of a detailed design. Because these products are needed to produce an accurate software plan, software planning must be an iterative process. Software planning is normally accomplished in three stages:

1) A *concept-oriented plan* is prepared based on a rough understanding of the system concepts (top-level requirements).

2) After the system requirements are clearly defined and well understood, the concept-oriented plan is modified and becomes the *capability-oriented plan.*

3) When the program detailed design is complete, the capability-oriented plan can be modified to become the *implementation-oriented plan.*

The implementation-oriented plan is then used during software production to track the project, with adjustments made to the plan as necessary. Figure 1 shows how these three plans fit in with the software development life cycle.

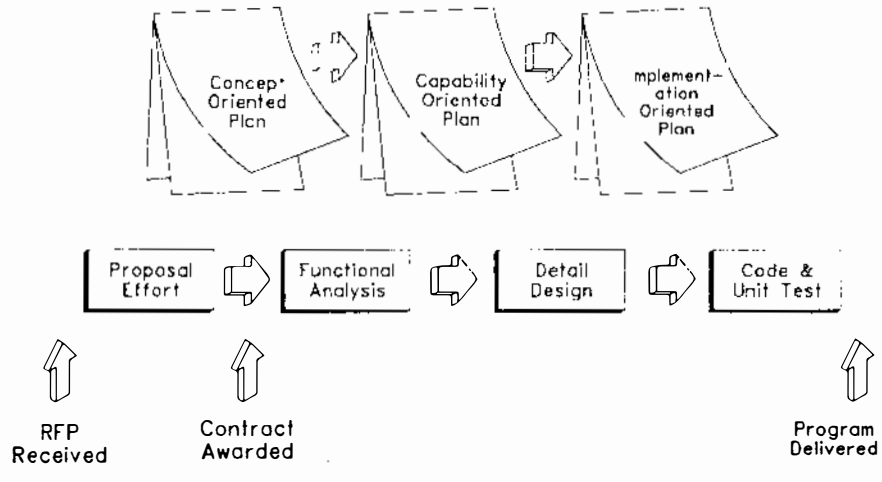


Figure 1

Concept-Oriented Plan: The concept oriented-plan can be prepared as soon as the rough system requirements are understood. The concept-oriented plan typically includes detailed information about all tasks through completion of the Program Functional Specification along with top-level descriptions of the detailed design, coding, and testing related tasks. The concept-oriented plan will, however, include all known major

milestones throughout the entire project. The concept-oriented plan is typically based 80% on inputs from the project management group and 20% on inputs from the project technical staff. Total system cost estimates at this stage are "ballpark figures" and can be expected to be accurate plus or minus 50%.

Capability-Oriented Plan: The capability-oriented plan revolves around the functional capabilities identified in the Program Functional Specification, with schedules and cost estimates based on these functional capabilities. Detailed time and cost planning for both design and coding stages are possible, although the task breakdowns for these areas are based on functional capabilities only. At this stage, rough functional and/or data element costing metrics become available, allowing total system cost estimates to be prepared which are accurate to plus or minus 25% The capability-oriented plan is typically based 50% on inputs from the project management group and 50% on inputs from the project technical staff.

Implementation-Oriented Plan: The implementation-oriented plan can be prepared after completion of the detailed program design and is based on coding, testing, and integration of specific program modules. For the first time, the plan will have a true one-to-one correlation with a physical unit of software. Finally, true functional and data element metrics are available, allowing total system cost estimates to be prepared which are accurate to plus or minus 10%. In addition, it is only at this point that reasonable use can be made of Lines of Code (LOC) based costing models. The implementation-oriented plan is typically based 20% on inputs from the project management group and 80% on inputs from the project technical staff.

The actual planning process during each of the three scheduling stages is identical. The steps required are as follows:

1) Decompose the tasks to be performed.

2) Define the dependency relations between the tasks.

3) Estimate the resources required to perform each task.

4) Perform a risk analysis for each task.

5) Schedule the project by scheduling all tasks to be performed.

We will now discuss each of these five areas in more detail.

Task Decomposition

Any project can be decomposed into a hierarchy of discrete events which describe the tasks to be accomplished. The resulting task breakdown will then serve as the basis of further planning and control. Specifically, task decomposition is important for the following reasons:

- Task decomposition breaks the project down into work packages with a high degree of granularity, thus facilitating project tracking and control. In addition,

time and cost estimates that are prepared based on many small work packages are much more accurate than estimates prepared based on top-level project requirements.

- Task decomposition allows you to assign responsibility for individual tasks and provides specific and verifiable measures of successful completion. If you are working with subcontractors, the task hierarchy provides a valuable input to preparing delivery orders or contract requirements.

- Decomposing the project into a task hierarchy helps to ensure that every major and minor activity required to complete the work will be accounted for.

- During the second and third planning stage (function-oriented plan and implementation-oriented plan), the task decomposition identifies interfaces and data flow requirements between functional areas or program modules.

- The task hierarchy will often provide visibility to specific areas of the project software which can be "captured" from previous work. For example, the function-oriented task hierarchy may identify functional areas which have been designed and coded on previous projects, allowing the entire design and code for these areas to be reused. The implementation (program module)-oriented hierarchy almost always identifies subroutines which can be captured, at least at the lowest levels.

Our goal is to decompose the project into a task hierarchy which defines all work required to satisfactorily complete the project and highlights all milestone events during the project. It is important to note that the tasks must be comprehensive - no work can be required which is not shown in our task hierarchy. Tasks at the lowest level in the hierarchy represent discrete work packages, while tasks at higher levels represent either integration and test efforts or logical groupings of related work. When decomposing the project, you should try to use the following general rules and guidelines:

- Tasks should be defined such that an objective method of evaluating task completion is available.

- Each element of work should be assigned to only one task, not spread among several locations in your task hierarchy.

- Each task must be clearly defined (in writing), or else only you will have a clear idea of what work has to be accomplished to complete the task.

- At the lowest level, tasks should be defined so that as few people as possible, preferably a single person, can be assigned responsibility for completion of the task.

- Tasks should have a time duration short enough to allow recovery in the event of total failure. As a rule of thumb, projects lasting less than six months should have tasks lasting 20 to 80 man-hours, projects lasting six months to two years should have tasks lasting 40 to 120 man-hours, and projects lasting over two years should have tasks lasting 40 to 240 man-hours.

- Tasks should be identified in a manner which maximizes task cohesion. (*Task cohesion* is a measure of how uniform the task's work is.) For example, work which includes hardware design and software development should normally not be included in the same task.

- Tasks should be identified to minimize task coupling. (*Task coupling* is a measure of how dependent one task is on other tasks for successful completion.)

HIPO-II (Hierarchy plus Input-Process-Output II) decomposes a software program into six classes of modules [Roetzheim, 1988b]:

1. *Menu Modules:* These modules represent menu choices available to the user.

2. *Interrupt Modules:* These modules represent processing initiated by an external interrupt.

3. *Keyboard Modules:* These modules represent processing initiated by a user key press (special function key, labeled key, etc.)

4. *Processing Modules:* These modules represent internal processing required to accomplish a specific task.

5. *Common Modules:* Common modules are processing modules which are fully defined somewhere else in the design hierarchy.

6. *Library Modules:* Library modules are identical to common modules except that the definition is external to this program design.

The Systems Analyst will typically begin by designing the program's menu, interrupt and keyboard modules. These modules fully define the program's user interface and functional requirements. This portion of the design hierarchy is then provided to the Project Manager as the primary input to the project capability-oriented plan. The Systems Analyst then adds processing, common and library modules to define internal processing requirements. The design hierarchy is then provided to the Project Manager and serves as the primary input to the implementation-oriented plan.

Defining Task Dependencies

All scheduling includes defining task dependencies, although this process is often performed automatically as you prepare the schedule. If all of our projects involved a relatively small number of tasks, the task dependencies could be easily kept in our head for use during scheduling. Unfortunately, many computer-related projects involve hundreds or thousands of tasks, preventing most people from developing the complex dependency definitions in an informal fashion. In addition, the planning for most large projects involves coordination with a project team. This coordination requires an effective method of communicating all aspects of the schedule, including dependencies - something which is not possible without formal dependency definitions.

The most common type of dependency we will be working with is the "finish-to-start" dependency. Task 1 must be fully complete before task 2 can be started. Other types of dependencies are possible, and some of these are quite common in the real world of software development. For example, if you had one task called "Code Software Modules" and a second task called "Integrate Software Modules," you would normally want to say that task 1 (code) must be partially complete prior to the start of task 2 (integrate). We call the task which must be started first the *predecessor task* and the task which must begin later the *successor task.* Some computer programs allow you to define any type of dependency desired by specifying a percentage complete for the predecessor task. Our software development tasks might then be defined by saying that task 1 (code) must be 25% complete before task 2 (integrate) can be started.

With an understanding of the basic concepts of dependency definitions under our belt, we are ready to look at the specifics of defining the dependencies for our project. On large projects involving hundreds or thousands of tasks, you might be tempted to throw in the towel when sitting down to define the task dependencies. Let me clue you in on four secrets which will keep this job manageable for even the most complex project:

1. Dependency links should be defined at the highest level possible in the task hierarchy. Suppose that program A must be complete prior to starting on program B, but individual modules within both programs can be coded in any order. We should show this as a single dependency link rather than showing every detail dependency.

2. Do not try to explicitly show implied dependencies. For example, suppose task A must be complete before Task B can be started, and task B must be complete before task C can be started. There is an implied dependeny link between task A and task C: task A clearly must be complete before task C starts. We need to show the links from A to B and B to C, but the implied link from A to C does not need to be shown.

3. Most computer programs allow you to describe all predecessor and successor tasks for a selected task. With a little thought, you will see that every predecessor link is a different task's successor link (and vice versa). This means that if you define *all* predecessor and successor links in a task hierarchy, one-half of your links will be redundant. For this reasons, it is possible to define only predecessor or only successor links throughout the network, and still completely define the task dependencies.

4. Do not try to use dependency links to force a logical schedule. In other words, avoid defining dependencies unless the link is significant. For tasks which have vague links such as "This task should be done early on in the project" or "This task should be started during the last quarter of the project," you should not try to force the schedule by defining fictitious dependencies.

Resource Estimating

For each stage of software planning, you must estimate the resource requirements for individual tasks identified during project decomposition. These resource estimates will allow you to estimate total resource requirements, and to determine resource requirements over time for the life of the project. One common misconception about resource estimates is that the initial estimates prepared during the concept-oriented plan are the only important estimates, and that resource estimation at later scheduling stages is a waste of time because "the project budget is already fixed in concrete." This ignores the critical fact that estimates prepared during later stages of software planning are *always* more accurate than earlier estimates. These later estimates can be used to validate earlier estimates, or to point out cost related problems in the project.

If your estimates point out a problem with your earlier cost estimates, you have three alternatives available to you:

1. You can ignore the later estimates (the equivalent of not doing the estimates). This alternative results in the common problem of cost overruns occurring at the later stages of the project and almost always results in a high degree of customer dissatisfaction.

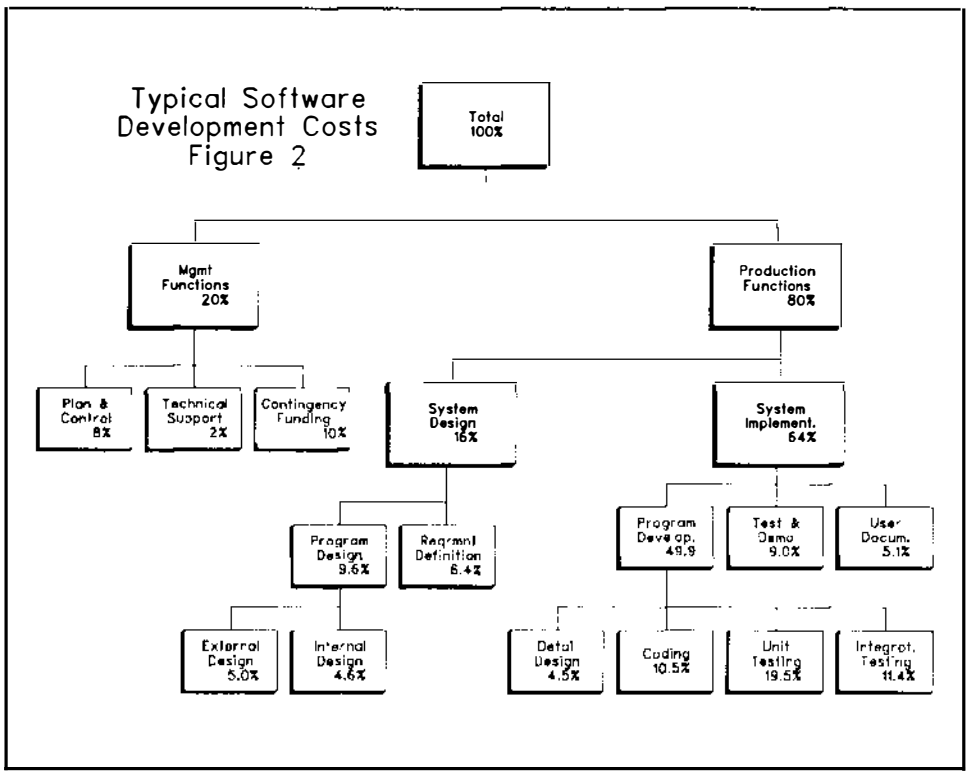
2. You can revise your earlier estimates and request additional funds from the customer. If the customer understands the fact that estimates prepared during the concept oriented planning stage are only accurate within plus or minus 50%, this option might be reasonable. In any event, it is always better for the customer to be apprised of this type of news as early as possible (as opposed to waiting until you actually do run out of money).

3. Finally, you can use the function- or module-based cost estimates to revise your design to match the original estimates. This "design to cost" approach is extremely effective under these circumstances both because the precise causes of high costs are clearly understood and because it is early enough in the project for you to do something about it.

Resource estimation for software development tasks is normally a three step process. First, the manpower required to perform the task is estimated. These manpower estimates are normally based on a category labor (senior programmer, systems analyst, etc.) and are expressed in man-hours or man-days. Manpower estimates normally assume an optimal task duration and are adjusted later if development is accelerated. Second, the calendar time required/available for each task is determined. This task duration will be used during task scheduling. In addition, there is a resource/cost penalty associated with task durations which are less than or greater than the optimal duration. Finally, the actual cost to complete the task is calculated using the adjusted man-hours for each labor category and the hourly cost for that category of labor. Specific algorithms for calculating optimal durations and determining cost penalties for accelerated development are described in detail in [Roetzheim, 1988a].

Let's talk about a quick and dirty method of estimating costs which is appropriate for rough estimates and for very small projects. The hierarchy shown in figure 2 shows the typical percentages for software development costs broken down by category of work.

The percentages are based on my experience, so feel free to adjust to your project environment.



Risk Analysis

Performing a detailed project risk analysis is useful to you during planning for the following reasons:

- Resources (time, money, and management attention) can be allocated based on each task's risk.
- Candidate tasks for prototyping can be identified. Prototyping risky software functions or modules is often the most powerful method available to you to reduce risk.
- A risk-optimized schedule can be generated. A risk-optimized schedule attempts to schedule risky tasks as early as possible in the project to allow sufficient time for recovery in the event of failure.
- Contingency planning is possible.

The amount of time spent on contingency planning is proportional to both the project's risk and priority. Some factors to look at when determining what amount of contingency planning is appropriate include:

1. The technical risks in development.

2. The financial or competitive consequences of failure or late delivery.

3. The value to your company of completing the project successfully. Value can include cost savings, profit increases, or return on investment.

4. The amount of influence that the customer possesses.

5. The impact of project failure on other projects, affiliated organizations, or the strategic direction of your company.

Software development projects typically have three types of risk associated with each task: (1) network risk, (2) technical risk, (3) schedule risk (which also relates to cost risk). For each of these types of task-related risk, we will want to determine the task's overall task risk factor (R_t). this will often involve first calculating the likelihood-of-failure factor factor (L_f) and consequence-of-failure factor (C_f). These numbers are expressed as integers from 1 to 99 and are combined using the formula:

$$R_t = L_f + C_f - .1(L_f * C_f)$$

In addition, a cumulative task risk factor consisting of the average of these four factors is normally computed. Many risk factors can be automatically calculated using a program such as *Structured Project Manager's Toolbox (SPMT)* [Roetzheim, 1988c].

Calculating network risk components. Network risk involves risk related to the dependency network linking various project tasks. Tasks which have a large number of predecessor tasks have a high network related likelihood-of-failure, while tasks with a large number of successor tasks have a high consequence-of-failure. SPMT uses a linear function based on the number of predecessor and successor tasks to calculate each task's likelihood-of-failure and consequence-of-failure, then averages these numbers to arrive at the task's network risk component.

Calculating technical risk components. A task's technical risk analysis looks at the risk associated with a failure to complete the task to a required degree of technical excellence. For example, a parallel port line driver which must operate at 9600 bytes per second would be a technical failure if the code could not be made to handle the appropriate data rates. Note that this would be a technical failure even if the module was completed on time and on budget. To calculate each task's technical risk factor, we must first calculate the technical likelihood-of-failure and consequence-of-failure factors.

A task's *technical likelihood-of-failure factor* is a number representing your estimate of how likely the task is to miss its technical goals, even with a reasonable cost or schedule overrun. The likelihood-of-failure factor for software development tasks is best estimated by independently examining the following four potential causes of technical failure and using the average of all four values:

1. The likelihood of failure due to the immaturity of the hardware technology

2. The likelihood of failure due to the immaturity of the software technology

3. The likelihood of failure due to the complexity of the system's hardware

4. The likelihood of failure due to the complexity of the application software.

A task's *technical consequence-of-failure factor* is a number which represents your estimate of how serious a task failure will be in a technical sense. We can then calculate the technical risk factor for each task by combining the likelihood-of-failure factor and the consequence-of-failure factor.

Calculating schedule risk components. Our final risk component deals with the risk that the task will require more resources than planned for, normally in terms of time or dollars. During the resource-estimation stage of the project planning, we determine our most likely estimate of each task's manpower requirements to complete the work. We now improve this estimate by including the following additional variables for each task:

- an optimistic estimate of the manpower required to complete the task
- A pessimistic estimate of the calendar days required to complete the task.

These numbers allow us to calculate the expected manpower requirements along with the standard deviation. Techniques described in [Roetzheim, 1988a] can then be used to calculate the optimal duration for this task, and for a given duration, the probability of not completing the task on time (the likelihood-of-failure factor), and the expected overrun (consequence-of-failure factor).

Scheduling the Project

A software schedule is measured in terms of its: 1. Time; 2. Cost, and; 3. Risk. Ideally, we would like to produce a schedule which is optimal for all three of these variables, but this is never possible. For example, you can produce a software development schedule which accelerates development tremendously (optimum in terms of time), but this will always cost more than development at a more orderly pace. It is possible to produce a schedule which is optimal in terms of any one of the variables, at the expense of the other two, but this is done only in unusual circumstances. For example, a crash program to produce software under emergency circumstances might trade high risk of failure and high cost for a potential delivery in record time.

On most projects, an acceptable limit for two of the variables is selected and the schedule is optimized in terms of the third variable while holding the other two at their acceptable value. Other variables which might act as limiting factors during software scheduling are:

- Required calendar completion dates
- Cash-flow restrictions
- Limited resources, especially people

- Required administrative time to obtain approvals

You should use the following guidelines during scheduling:

1. Be sure that you do not schedule tasks in conflict with your defined dependency definitions.

2. You should not schedule tasks to require more resources during any given month than will be available. Resources might include terminal hours, specific individuals or categories of labor, or cash.

3. You should minimize parallel assignments of personnel. For most activities people work most efficiently when they can concentrate on one task at a time.

4. You should make maximum use of resource leveling. *Resource leveling* is defined as moving tasks to minimize fluctuations from week to week in required resource levels. It is obviously preferable to have ten programmers working on the project from week 1 through week 20 rather than having 30 programmers required for the first five weeks, none for the next ten weeks, and 10 programmers required for the last five weeks.

5. You should maximize resource cohesion. *Resource cohesion* is a measure of the uniformity of the type of work individual project team members are assigned to work on during the project.

Summary

Project management involves more activities than simply monitoring software development, including pre-development activities such as analysis and evaluation of the contract prior to bid and marketing along with post-development activities such as post completion evaluation. Because project planning is required throughout the project management, often well before a software detailed design is started, an iterative approach to planning is necessary. Typical projects involve three stages of planning:

1. Concept oriented planning prior to contract award.

2. Capability oriented planning prior to completion of program design.

3. Implementaton oriented planning after completion of the program design.

A software design technique called HIPO-II supports this iterative type of planning by supporting multiple classes of software modules. *Menu, Interrupt,* and *Keyboard* modules are primarily used during concept and capability oriented planning to support functional decomposition of the problem. *Processing, Common,* and *Library* modules are then used during implementation oriented planning to represent the actual software design.

Finally, the topics of resource estimating, risk analysis, and project scheduling were dealt with briefly.

References

Roetzheim, W.H., *Structured Computer Project Management*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988a

Roetzheim, W.H., *Structured Design Using HIPO-II*, Englewood Cliffs, NJ: Prentice-Hall, Inc., estimated November, 1988b

Roetzheim, W.H., *Structured Project Manager's Toolbox User's Manual*, San Diego, CA: William H. Roetzheim & Associates, 1988c

Roetzheim, W.H., *Proposal Writing for the Data Processing Consultant*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986

Project Management 3.

"Software Developer and Vendor Liability"
Nancy E. Willard, Attorney at Law

"FI^{3}T: An Evolutionary Approach to Software Development at Mentor Graphics"
Rick Combest and Sue Strater, Mentor Graphics

"A Methodology for Software Maintenance"
John E. Moore, Hercules Aerospace

SOFTWARE DEVELOPER AND VENDOR LIABILITY

Nancy E. Willard
Attorney at Law
296 E. 5th Avenue, Suite 302
Eugene, Oregon 97401
(503) 342-3599

ABSTRACT

An increasing number of software developers and vendors are finding themselves facing lawsuits or threats of lawsuits from dissatisfied users. The situations which are leading to litigation include misrepresentation of the capabilities, software failure and failure to deliver software as promised. This paper presents an overview of some of the key issues of software litigation and suggestions on how a software company can avoid or limit the potential damaging effects of litigation.

BIOGRAPHICAL INFORMATION

Nancy E. Willard is an attorney who practices in the area of computer law and technology transfer. Ms. Willard is actively involved in economic development activites for science and technology industries in the state of Oregon. She was recently appointed by Governor Neil Goldschmidt to the Oregon Software Industry Development Council. She is the past president of the Eugene Software Council and is a current member of the Coumpter Law Association.

Ms. Willard has presented a number of lectures to the software industry on legal aspects which affect the industry.

SOFTWARE DEVELOPER AND VENDOR LIABILITY

Nancy E. Willard
Attorney at Law
296 E. 5th Avenue, Suite 302
Eugene, Oregon 97401
(503) 342-3599

OVERVIEW

An increasing number of software developers and vendors are finding themselves facing lawsuits or threats of lawsuits from dissatisfied users. Due to the inherent sluggishness of the judicial system, not all of the questions or issues that arise in this area have been clearly resolved. This paper will present an overview of some of the key issues of software litigation and will present suggestions on how a software company can avoid or limit the potential damaging effects of litigation.

Software litigation results from a variety of problem situations. Typical problem situations include:

1) Misrepresentation of the capabilities of the software, resulting in failure of the software to meet the needs of the user.

2) Failure of the software to perform properly, causing economic injury to the user.

3) Failure of the software to perform properly, causing personal injury or damage to property.

4) Failure to deliver the software in accordance with an agreed development schedule.

These lawsuits are brought on a variety of theories, including contract theories of breach of contract and breach of express and implied warranties and tort theories of professional malpractice, mispreresentation, fraud, negligence, and strict liability. The attorney for the user will plead as many claims for relief as possible, even if the claims may be based on exactly the same facts. Because this area of litigation is relatively new and the case law is still unfolding, pleading several claims allows the user to preserve alternative claims for relief against the possibility that one or more claims may not be upheld.

An attorney reviewing a potential software liability case will consider four basic questions:

1) What are the possible legal theories or causes of action that would support a claim for damages?

2) What defenses could the software company raise to these claims?

3) What damages would be recoverable?

4) What facts must be established by the plaintiff and to what degree of certainty?

LEGAL THEORIES FOR RECOVERY

Contract Claims

The initial question that arises in a contract situation is whether Article 2 of the Uniform Commercial Code (UCC) would govern the situation. UCC governs the sales of goods. The purpose of the UCC is to clarify and modernize laws governing commercial transactions, to provide sufficient flexibility to enable continued expansion of the law to address new situations and to promote uniformity of laws across state jurisdictions.[1] The Code has been adopted by all states, with the exception of Louisiana. Because vendors, courts, and attorneys are familiar with the UCC and rely on its predictability, there is a strong tendency to favor its application in a given situation.

Many software transactions involve aspects of both sale (or license which is considered equivalent) and service. To determine whether the UCC will govern in software transactions requires an analysis of the "predominant feature" of the transaction.[2] When the UCC does not govern a situation, the common law of contracts will govern. In many situations, this is a distinction without a difference because the UCC is a codification of the common law.[3]

An analysis of recent case law reveals the following trends.[4] If the transaction involves the sale or license of standardized software with or without computer hardware, the courts consider the transaction a sale of goods under the UCC. If the transaction involves the sale of hardware and development of custom software, the courts tend to consider the software development as ancillary to the hardware transaction and, again, apply the UCC. If the transaction merely involves custom programming services, the courts would generally considered the transaction to be a services contract and outside the scope of UCC.

Under either the UCC or common law contracts, the plaintiff must plead and prove the following elements: 1) the existence of a valid contract, 2) the plaintiff's performance or excuse for non-performance of the contract;

3) the defendant's breach and 4) remedies to which the plaintiff is entitled.[5]

A valid contract requires an offer, acceptance of that offer, and consideration which is something of value given or promised by both parties.

The plaintiff must be be free from substantial default under the terms of the contract. For example, in a custom programming situation, if the plaintiff is required to provide certain information or to perform certain evaluations during the course of the development and fails to do so, any breach of contract by the defendant may be excused.

A breach of the contract occurs when there is a wrongful, that is, unjustified and unexcused, failure to perform the terms of the contract. Under common law, any failure to conform to the provisions of the agreement, including failure to deliver software in a timely manner and failure of the software to perform properly, could be considered a breach.

A breach of contract may involve a breach of one or more warranties which arise under the UCC. These are an express warranty, an implied warranty of merchantability and an implied warranty of fitness for particular purpose.

An express warranty is created by any affirmation of fact or promise relating to the goods, any description of the goods or any sample or model.[6] There is no requirement that the seller intend to create an express warranty. It is necessary to distinguish between express warranties and mere "sales puffing". The Code disfavors attempts to exclude or disclaim express warranties. Permitting disclaimers would allow vendors to promise the moon to make a sale, while keeping their fingers crossed to ward off any liability.

The two implied warranties will arise automatically under certain circumstances unless they are carefully disclaimed. An implied warranty of merchantability requires that the goods pass without objection in the particular trade involved or be fit for ordinary purposes for which the goods are used.[7] An implied warranty of fitness for purpose arises when the vendor has reason to know of any particular purpose for which the goods are required and that the buyer is relying on the vendor's skill or judgment in making a particular selection.[8]

The remedies which are available upon the breach of a contract are intended to place the plaintiff in the position it would have been in had the defendant not breached the contract.[9] Damages may include actual damages, such as,

the purchase price of the software, and incidental and consequential damages.[10] Incidental damages are damages incident to the breach, including such items as storage and shipping costs. Consequential damages are losses which are incurred by the plaintiff as a consequence of the breach by defendant. Consequential damages must not only be a natural result of the breach but they must also have been reasonable foreseeable by the parties at the time they entered into the contract. Because the failure of a computer system can have significant financial consequences in certain business settings the question of the availability of consequential damages is crucial in software litigation.

Software companies generally attempt to either disclaim express and implied warranties or limit them to a certain period of time. It is also standard to disclaim any liability for incidental and consequential damages. Such disclaimers or limitations are allowed under the UCC.[11] In software litigation such attempted limitations or disclaimers appear to be effective only if they are reasonable and fair under the circumstances. If the effect of a disclaimer or limitation is to place an unfair burden on the user, courts appear willing to find ways to void or avoid such provisions.

The courts have used a number of methods to void or avoid a limitation or disclaimer which it considers to be unfair. In Hawaiian Telephone v. Microform Data Systems, the court found that the disclaimer of implied warranties and limitation of consequential damages applied only to the software program upon final delivery, in this case, the developer had failed to deliver, therefore the disclaimer and limitation did not apply.[12] In RRX Industries v. Lab-Con, Inc., the court determined that since the developer promised to provide a system that worked or remedy any defects and was unable to do so, the remedy had failed of its essential purpose.[13] Because the remedy failed, the limitation of damages was not enforced.

The courts may also refuse to enforce provisions which limit remedies or disclaim warranties on the basis that the effect of enforcing such provision would be unconscionable.[14] The enforcement of such provisions is considered unconscionable in situations where the effect of the provision is harsh and where there has been either: 1) unfair surprise, such as deceptive sales practices or hidden contractual provisions, or 2) in situations where one party has a superior bargaining power. Many software licenses are offered on a "take-it-or-leave-it" basis, to users who do not have significant technocal expertise. If the effect of limiting or disclaimer provisions is harsh, such provisions could easily be found to be unconscionable. It is much less likely that a provision in a contract that has been

negotiated between two knowledgable business parties would be found to be unconscionable.

Tort Claims

As noted above, because the case law in this area is still evolving and because of a desire to avoid certain unfavorable contractual limitations on warranties and damages, plaintiff's attorneys also frequently raise tort claims in software litigation cases.

The trend is to disfavor tort claims, except in cases of injury to person or property and fraud.[15] Claims of professional malpractice and negligent misrepresentation are probably not well supported under current legal standards.

Professional malpractice is a claim that is difficult to substantiate at this time. The definition of a professional is one who has "a special form of competence which is not shared by the average reasonable man, but which is the result of acquired learning and aptitude and is developed by special training and experience."[16] Professionals are held to a higher standard of care. Whether software developers are professionals in the legal sense is still an open question.[17]

Negligent mispreresentation, as distinguished from intentional misrepresentation or fraud, is generally raised in an attempt to avoid the negative consequences of the disclaimers of warranty or limitations of remedies under the UCC.[18]

The elements of fraud or intentional misrepresentation are: 1) a false or misleading misrepresentation or omission made by the defendant, 2) knowledge by the defendant that the representation was false; 3) that the plaintiff rely on the misrepresentation, 4) justifiable reliance by the plaintiff on the misrepresentation and 5) damages resulting from plaintiff's reliance.[19] In most jurisdictions fraud must be proven by clear and convincing evidence, a higher burden than standard negligence.

Because there is no statutory basis for fraud, the legal standards vary according to the jurisdiction. Plaintiffs have an affirmative duty to mitigate damages however the extent of this duty will depend on the facts of the particular case.[20] In most jurisdictions fraudulent representations made either as part of a contract or in the precontract negotiations provide a basis to invalidate provisions limiting warranties or damages.[21]

Claims of negligence or strict liability are raised in situations where the use of a software product has caused an injury to person or property.

The elements necessary for a claim of negligence are: 1) the existence of a duty on the part of the defendant to protect the plaintiff from injury, 2) breach of that duty by act or omission to act, 3) a causal connection between the breach and plaintiff's injury, and 4) actual damages. [22]

The standard of care required of a defendant is the exercise of ordinary or reasonable care, that is, the conduct of an ordinarily or reasonably prudent person in like circumstances.[23] The amount of care and kind of conduct will vary with the circumstances but the standard does not vary.

Generally, the defendant owes a duty of care to all persons who are forseeably endangered by its conduct. A breach of duty by the defendant is an act or omission that exposed the plaintiff to an unreasonable risk of harm.[24]

The defendant's negligence must have been the cause-in-fact of plaintiff's injury, that is, "but for" the defendant's negligence, plaintiff's injury would not have occurred. Defendant's conduct must also be the proximate cause of plaintiff's injury, that is, plaintiff's injury must have been a foreseeable result of the defendant's negligent act.[25]

The acts or omissions of the plaintiff or other defendants may have contributed to the injury to plaintiff. Such contributory negligence may reduce or eliminate the damages assessed against any one individual defendant.[26]

The plaintiff is entitled to recover all damages proximately caused by his or her injury. These include personal injuries, property damages, and, in some states, pure economic loss. However, the plaintiff must act reasonably to mitigate any damages and if the plaintiff fails to do so the amount of damages may be reduced.[27]

Strict liability is liability without fault. A manufacturer or vendor can be held strictly liable for damages which may occur when a product is placed on the market with the knowledge that it will be used without inspection and it has a defect which causes injury to prson or property. Strict liability would not apply to services, such as a custom programming situation.[28]

Strict liability is imposed for several reasons: 1) the manufacturer or vendor is better able to insure against the loss than the individual consumer, 2) the belief that manufacturers will use more care if they can be held strictly liable, and 3) the potential difficulty of proving a manufacturer's negligence, particulary when the product is complex.[29]

The elements of a strict liability claim are: 1) the product had a defect when it was sold or leased to the consumer, 2) the product was being used in a normal, intended, or reasonable foreseeable manner when it caused the injury, and 3) the defect was the proximate cause of the injury.[30] There is no requirement that the plaintiff was the purchaser of the product. Damages may include personal injury and property damage.

There are two basic kinds of product defects that can lead to strict liability; 1) manufacturing defects, where the product deviates from the manufacturer's intended result, and 2) design defects, where the product fails to perform safely and the risk inherent in the product's design outweighs the benefit of the design. The defendant bears the burden of proving the benefit of the design outweighs its risks.[31]

BURDEN OF PROOF

The plaintiff generally bears the burden of proof in both tort and contract cases. With the exception of fraud, the level of proof is the preponderance of the evidence, that is, it is more likely than not that a certain fact is true. In certain situations, however, the burden of proof may shift from the plaintiff to the defendant. In general, this shift of the burden occurs in situations where the plaintiff has obviously suffered a loss but evidence of the cause of that loss is difficult for the plaintiff the obtain.[32]

It is readily apparent that a plaintiff would bear an exceptionally heavy burden if it were responsible for identifying programming errors or machine malfunctions. In a multi-vendor situation, it would be extremely difficult for the plaintiff to identify which vendor bears the responsibility for the loss. It would be unfair to require the plaintiff to prove all of the elements of its case, when its loss was obviously caused by the defendant or defendants, especially when the defendant or defendants have better access to the information of how the loss came about.[33]

Authorities differ on when the plaintiff's burden of proof in a software liability case is satisfied, however, the clear trend appears to be to only require the plaintiff to establish either nonperformance or faulty performance. The defendant would then bear the burden of proving that its actions were not the cause of the failure.[34] In multi-vendor situations, the burden would shift to the individual defendants to establish which defendant was responsible for the nonperformance or faulty performance.[35]

PRACTICAL SUGGESTIONS

As one author recently noted,

> "Modern litigation, like nuclear war, is a frightening prospect. Its potential victims should seek more peaceful alternatives. Whether due to liberal discovery, an adversarial and litigous culture, clogged courts, the modern law firm with its extensive resources and eager associates, or simply the complexity of modern transactions, litigation can be expensive, slow, petty, time-consuming, unfathonable, capricious and just plain exasperating.[36]

Although the law is far from settled, software developers and vendors can take positive steps to limit their potential liability. These positive steps include:

1) Establish effective project controls. Insure that you and your employees are keeping up with state-of-the-art-software technology, specifically software quality and testing technology. Test your software effectively and thoroughly following (exceeding) industry standards for software quality control. Insure that your software is user-friendly and that your documentation is clearly understandable.

2) Thoroughly document everything, the design and development process, the testing process, problems which are discovered in the product after release, your remedial actions, and interactions with customers (especially in problem situations).

3) Insure that the statements about your product or services made in your literature are accurate. Maintain careful control over anyone representing your product to the consumer to insure that they are not overselling or misrepresenting your product to the consumer.

4) Pay careful attention to contractual provisions, specifically those which address the responsibilities of the parties and the burden of certain risks. Make sure that the language of your contract is clear, understandable and unambiguious and that the agreement, particularly the allocation of the burdens of risk, is fair under the circumstances. To avoid claims of unfair surprise, make sure that your customer actually reads the contract. Watch out for the effect of pre-contract documents which may have established express

warranties or other requirements. Insure that any amendment made to the contract is in writing and signed by both parties. Contractually provide for a process of mediation for the resolution of any problems, followed by arbitration, if necesssary, instead of litigation.

5) In custom development situations, communicate frequently with the customer and insure that the customer is involved in and approves every step of the process. In conjunction with the customer, prepare specific functional specifications, a detailed project timetable and a comprehensive acceptance testing procedure for both performance and reliability. If the customer fails to perform its obligations, document such failure in writing to the customer.

6) Emphasize to your customers the need to maintain equipment properly and to make a practice of routinely backing-up data files. Provide the availability of customer training for your product. Insist on providing training in situations where the use of the software requires some sophistication.

7) Establish an emergency response team to handle any software failure situations. In the event of significant failure or significant losses, obtain an independent expert analysis of the situation at the outset.

8) Know your limits. Don't take on projects which you are not competent to handle and don't make promises you can't keep.

REFERENCES

1. UCC 102-2.
2. See eg. RRX Industries v. Lab-Com, 722 F2d 543 (9 Cir 1985).
3. See eg. Data Processing Services, Inc. v. L.H. Smith Oil Company, 492 NE2d 314 (Ind Ct App 1986) (Trial court decided case under UCC, appellate court upheld award of damages but based decision on common law.).
4. Rodau, Computer Software Contracts: A Review of the Case Law, 21 Akron L Rev 45 (1987).
5. Scott, Commercial User-Vendor Litigation: The User's Point of View, 5 Computer L J 287, 289 (1985).
6. UCC 2-313.
7. UCC 2-314.
8. UCC 2.315.
9. Restatement 2d of Contracts, Sec. 347 (1981).
10. UCC 2-714, 2-715.
11. UCC 2-316, 2-719.
12. 9th Cir, October 9, 1987.
13. Supra, note 2.
14. UCC 2-302.
15. Kearney, Computer Dissatisfaction: Should Tort Remedies Be Permitted or Does the UCC Still Govern, 7 J of L & C 243 (1987).
16. Restatement 2d of Torts, Sec 299A, comment a (1965).
17. See, Chatlos Sys. Inc v. NCR, Co., 479 F.Supp 738 (DNJ 1979), modified, 635 F2d 1681 (3cir 1980), Cert dismissed, 457 US 112 (1982). (Court refused to apply computer malpractice theory.).
18. Kearney, supra, note 15 at 245.
19. W. Prosser, Law of Torts (4th ed. 1971) 685-6.
20. Scott, supra, note 5, at 321.
21. Id. at 322.
22. W. Prosser, supra, note 19 at 143.
23. Supra, note 16 at Sec. 283.
24. W. Prosser, supra, note 19 at 144.
25. Id.
26. Supra, note 16 at Sec. 463.
27. W. Prosser, supra, note 19.
28. Supra, note 16 at Sec. 402A.
29. Hall, Strict Products Liability and Computer Software: Caveat Vendor, 4 Computer L J 373 (1983).
30. Scott, supra, note 5 at 314.
31. Id.
32. Freidman & Siegel, From Flour Barrel to Computer Systems, 14 Rut C & T L J 289 (1988).
33. Id. at 294.
34. Carl Beasley Ford, Inc. v. Borroughs, Corp, 361 FSupp 325, 331 (ED Pa 1973), a'ffd 493 F2d 1400 (3 Cir 19740.
35. Freidman, supra, note 32 at 297.
36. Davidson, Project Comtrols in Computer Contracting, 4 Computer L J 133, 147 (1983).

FI^3T

An Evolutionary Approach to Software Development at Mentor Graphics

Rick Combest, APD Release Project Manager
Sue Strater, APD QA manager

(503)626-7000
Email: ogcvax!sequent!mntgfx!pdx!rickc or sstrater

Abstract:

An incremental development methodology named FI^3T has been defined and is currently in use on projects within Mentor Graphics Corporation. FI^3T is an acronym which stands for Functional, Incremental, Iterative, Integration, and Test. The term was coined because it "fit" the way the development process occurs naturally. The FI^3T process begins after requirements definition and prior to final system level test execution. Functionality is incrementally specified, designed, coded and tested during cycles, a baseline system is created at the end of the cycle which is then given to our client base to evaluate. Our cycles run in 4-6 week time periods.

Biographical Sketches:

Rick Combest graduated with a BS in Biology from the University of Oregon in 1976. He has worked as an engineer and manager on a variety of projects including business applications, real-time embedded systems, engineering applications and user interface management systems. Rick is currently performing Project Management and Release Management at Mentor Graphics.

Sue Strater graduated with a BS in Computer Science from Purdue University in 1983. She worked as an evaluation engineer on the iRMX real-time operating system product at Intel Corporation for 2 years before joining Mentor Graphics Corporation. Currently, Sue is QA manager of the APD division at Mentor Graphics.

1. Introduction

The traditional model of the software life cycle is the waterfall model. It defines a manufacturing oriented process of delivering software products. Interim deliverables occur at specific checkpoints to ensure that the product meets user and market requirements with appropriate quality and within budget and schedule constraints. The purpose of this process is to provide a means of determining accountability, tracking progress and assessing risk.

The waterfall model was developed in the 1970's under a well defined set of assumptions. It was assumed in this process that all product requirements could be understood at the beginning of the development phase. This implied that the "what" or functional aspect of a software product could be defined independent of the "how" or implementation aspect. These assumptions were driven from the constraint that machine resources were scarce and very expensive. In other words you better have it right by the time you design and code or the project was sure to meet disastrous consequences.

This model has served us well in the capacity for which it was designed. However, we have seen that for software products this model gives little insight into what actually occurs during the design and implementation phases of software development. Our experience has shown that often times we were caught unaware of development problems until the projected delivery date of a major milestone had come and gone without completion. It is at this point that we found our management skills could do little more than limit the amount of damage done to our project and all dependent projects. The process gave us no indication of impending disaster.

In addition we have seen that the original assumptions upon which the traditional model is based are no longer valid. Machine resources are no longer the throttling factor. The opposite is true today; we have an abundance of machine resources which invite us to find better ways to apply them in the software development process.

We have also experienced a geometric advancement in the base technology upon which we build our software products. This has served to invalidate the contention that all requirements can be defined before development begins and gives cause to suspect the notion that the "what" can indeed be separated from the "how".

What we need is a model which can retain the strengths of the traditional model to address the manufacturing and management control aspects of the software life cycle yet provides a process which accepts the fact that software development is a problem solving process. We need a model which can adapt to changes which occur due to requirement refinements, advancing technology, feedback from users or revelations of better ways of doing things. What we need is a model which software developers find natural yet defines a process that gives managers more insight into the software development process so as to allow management skills to be applied before a problem becomes a disaster.

There is at least one model which meets these requirements. This model is called Evolutionary Development.

1.1. Evolutionary Development

The Evolutionary Development process begins after the initial marketing requirements and product approval phases have completed and continues through to the delivery of a product for final evaluation. It fits inside traditional life cycle milestones.

Evolutionary Development presumes that a software product is grown in small steps or cycles, incrementally. The process produces real results for real users at each cycle. This is in contrast to prototyping which produces results which may or may not be real and often times these results are discarded.

Evolutionary Development is divided into cycles which are characterized by the following elements:

- Activities or tasks to be performed during the cycle. This includes such things as interviewing users, prototyping the user interface or designing and testing a specified piece of functionality.

- Intermediate products which are of product quality but may lack full functionality specified for the final product. Examples of this may be user interfaces with stubbed-out functions or utilities with interfaces partially complete.

- Control points which serve to monitor the planned versus actual progress of development. This may include reviews of designs or code, demonstrations of functionality or client reviews of intermediate products.

- Baselines of the system as it proceeds from cycle to cycle. In other words sets of intermediate products constituting a controlled version of the end product.

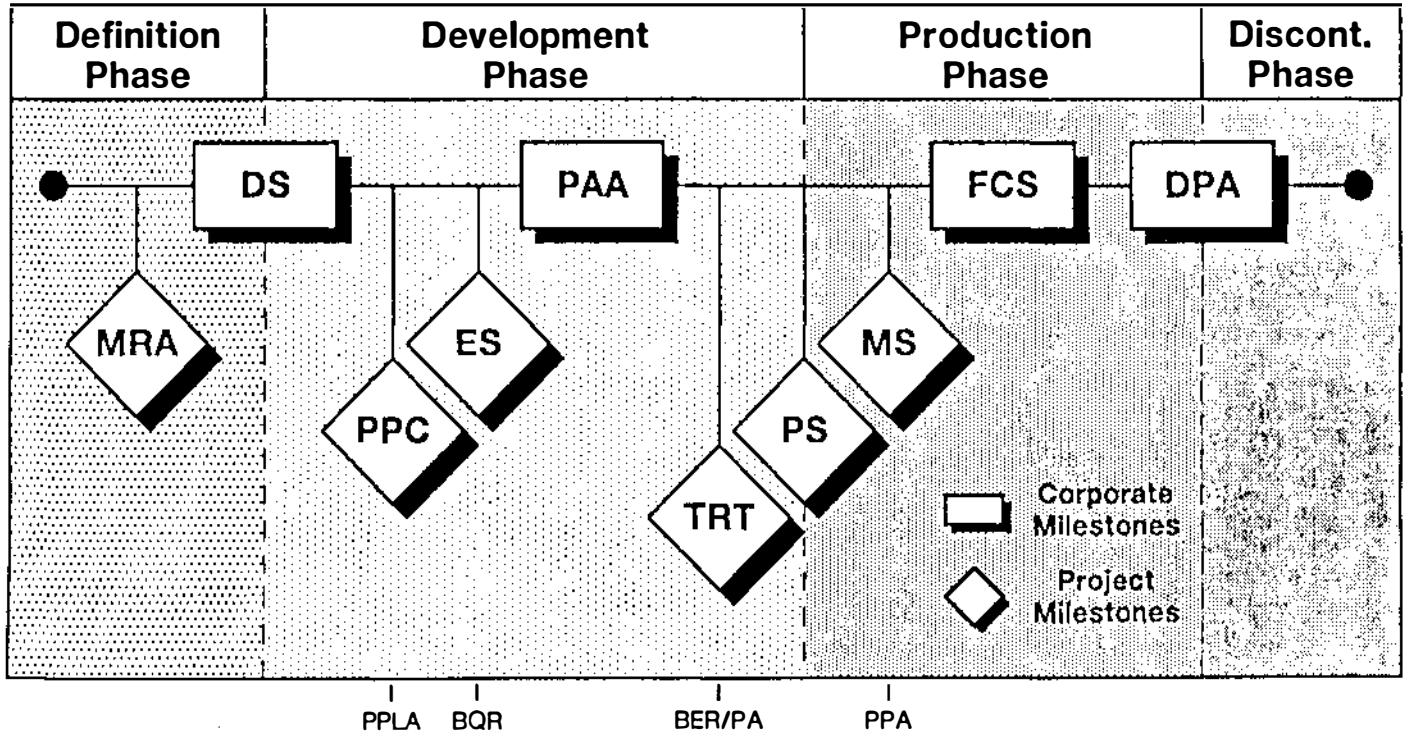
1.2. Product Life Cycle

Mentor Graphics has a corporate-wide product life cycle which is a tool for managing our product development projects. It was defined to improve quality, increase process productivity and improve profitability.

The Model includes corporate and project milestones which are shown in the diagram on the following page.

Mentor Graphics®

PRODUCT LIFE CYCLE



Corporate Milestones	Project Milestones	
DS – Development Start	MRA – Marketing Requirements Approval	PS – Pilot Start
PAA – Product Announcement Approval	PPC – Project Planning Complete	MS – Manufacturing Start
FCS – First Customer Ship	ES – Evaluation Start	
DPA – Discontinue Product Approval	TRT – Transfer to Release Team	

23 Nov 87

1.2. FI^{3}T

FI^{3}T is an instantiation of Evolutionary Development specifically tailored for Mentor Graphics Advanced Products Division. It is an acronym to describe the development phase of the PLC Life Cycle Model from Project Plan Complete (PPC) to Evaluation Start (ES).

FI^{3}T stands for:

- » FUNCTIONAL development process for Object Oriented Programming.
- » INCREMENTALLY develop functionality during cycles.
- » ITERATIONS occur due to requirements changing or learning better ways to do things.
- » INTEGRATE on a regular and frequent basis. The key to evolutionary method is the efficient building and releasing of new baseline systems.
- » TEST as you code. Testing occurs throughout the project and is retained. Problems in design, implementation or performance can be discovered and fixed early.

2. Case Study Description

We will be using a large project now in progress here at Mentor Graphics as a case study of the FI^{3}T process in a production environment. This project involves the delivery of two major subsystems, user interface and data base management, as well as a set of utilities and a development environment to internal clients. In parallel projects the clients will develop applications on top of the subsystems for delivery to customers. All developers will use an object oriented language and Apollo DSEE source code control system.

2.1. Detailed Process Description

The process of determining and documenting the overall system requirements will not change with the FI^{3}T methodology. System architecture and functional partitioning will occur at the beginning of development and will be refined as the requirements become clearer and better understood.

After high-level requirements and system partitioning are complete a Project Plan and accompanying Support Plans will be written to set project objectives and guide the project's overall development.

An initial Functional Specification for each major partition will be written which clearly documents the functionality needed to support the documented requirements in as much detail as is known at that time.

The initial Functional Specification, Project Plan and Support Plans will then be reviewed and approved. At the conclusion of this milestone, Project Plan Complete (PPC), the FI^{3}T process will begin.

The time between Project Plan Complete (PPC) and Evaluation Start (ES) will be divided into cycles. Each cycle will be planned to deliver pieces of the overall functionality specified during the planning phase. The cycle products will be new baselines of the overall system staged to meet client priorities for maximum parallel development efficiency.

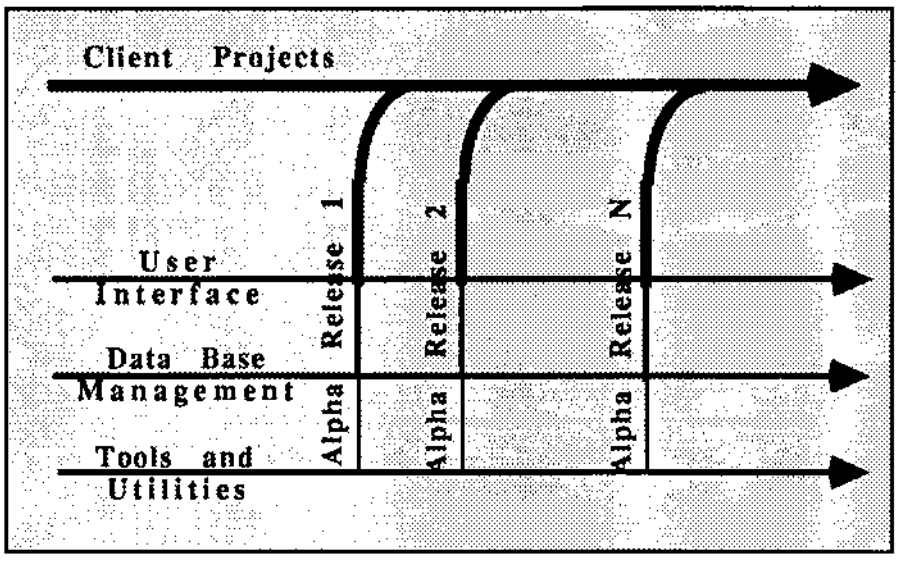


Figure 1 - Cycles in a Project Framework

Each cycle will include the following:

- A planning phase which produces a plan describing in detail what functionality is to be delivered at the completion of the cycle.
- A specification phase which produces an updated Functional Specification describing the functionality to be delivered in this cycle.
- A phase for design, code and test.
- An integration phase where all major system components are brought together.
- An evaluation phase where system tests are run to determine overall quality, performance and functional correctness.

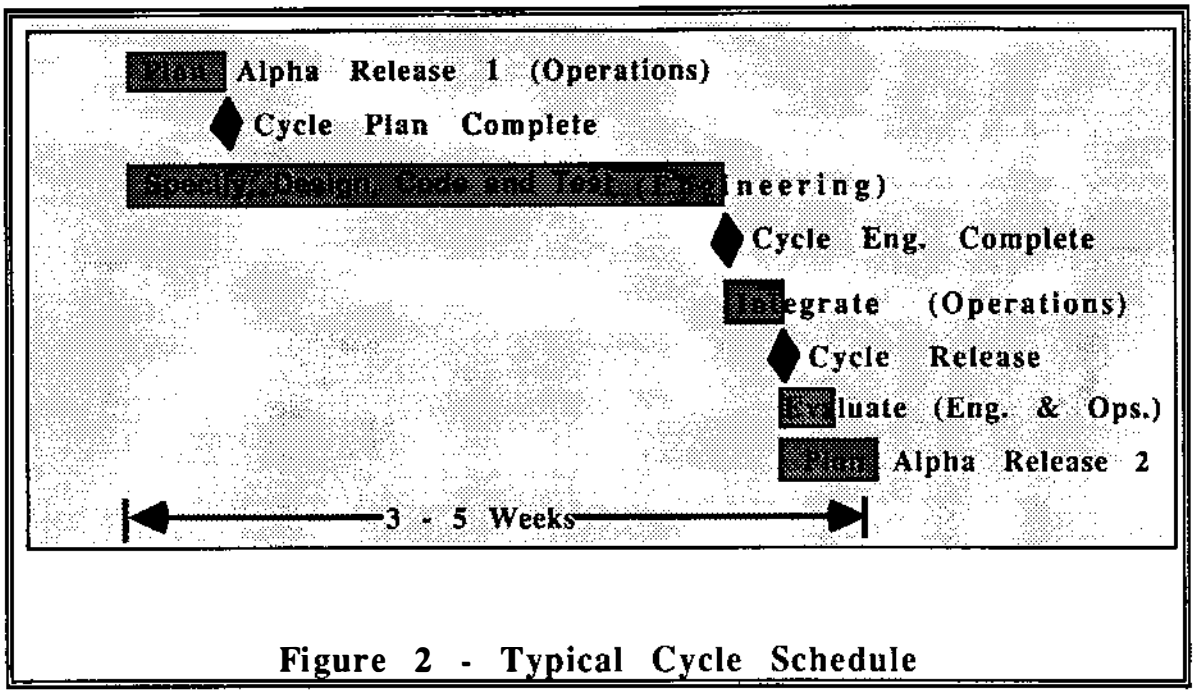
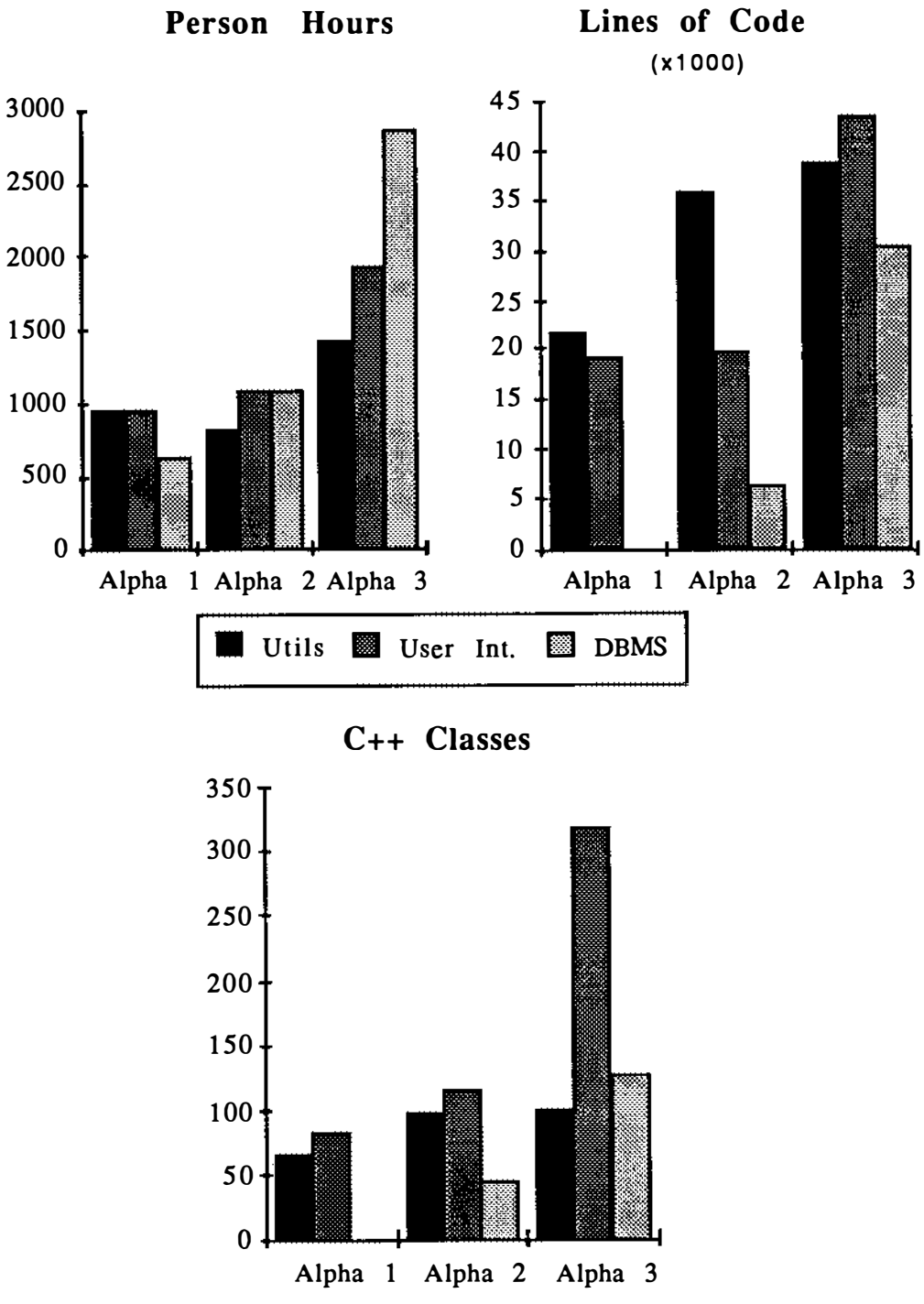


Figure 2 - Typical Cycle Schedule

3.0 Case Study Results

Development on this project is still in progress. We have completed three releases with two more scheduled this year. Measurements being tracked for each cycle include lines of code written, number of new C++ classes, man hours, number of reviews, and problems/enhancement counts reported by client base. A summary of some of the project data collected thus far is shown on the following page.


Person Hours
3000
2500
2000
1500
1000
500
0
Alpha 1
Alpha 2
Alpha 3
Lines of Code
(x1000)
45
40
35
30
25
20
15
10
5
0
Alpha 1
Alpha 2
Alpha 3
Utils
User Int.
DBMS
C++ Classes
350
300
250
200
150
100
50
0
Alpha 1
Alpha 2
Alpha 3

Data collected on problem reports and enhancement requests has not accurately reflected the relative quality of the delivered products. An informal process for communicating this information between project teams and the client base has distorted the results. If we are to use these measures from future releases we must find a way of quantifying exchanges in this informal process.

4.0 Summary

Benefits:

- Fits object oriented design methodology better than conventional waterfall models.
- Better adapted to changing or fuzzy requirements and base technology.
- Allows for the parallel development of projects by providing intermediate products at regular release points.
- Allows client feedback much earlier in the development phase. Changes can be made earlier at a much lower cost.
- Allows for earlier testing and performance gathering.
- Increases client base communication and satisfaction.
- Documentation stays up-to-date with the code.
- Provides for regular reviews during release cycles.

Difficulties:

- Dynamic nature of the process requires the use of a source code and configuration management system which can adapt to the evolutionary process.
- Client base must have a clear understanding of the nature of the software delivered with alpha releases. Especially important are expectations of relative quality, completeness and the need for timely feedback.

- Planning is critical. Contents of releases must be communicated to client groups efficiently.

- Must maintain discipline in functional testing during cycles. Release of unstable software leads distrust and frustration in the client base.

Conclusion:

FI^3T has allowed us to adapt to an extremely high rate of change in technology, requirements and even the process itself. Concluding that this process can be effective in large project development will have to await the completion of the project. However, the impressive results thus far have demonstrated success in maintaining high productivity in a parallel development environment.

REFERENCES

[1] Hekmatpour S. "Experience with Evolutionary Prototyping in a Large Software Project," *ACM SIGSOFT Software Engineering Notes,* 12(1), 38-41 (1987)

[2] Agresti W. W. "The Conventional Life Cycle Model: It's Evolution and Assumptions," *New Paradigms for Software Development,* IEEE Computer Society Press, 2-5 (1986)

[3] Brooks F. P. "No Silver Bullet, Essence and Accidents of Software Engineering," *IEEE Computer*, April, 10-19 (1987)

[4] Agresti W. W. "What Are the New Paradigms?," *New Paradigms for Software Development,* IEEE Computer Society Press, 2-5 (1986)

4] Agresti W. W. "Framework for a Flexible Development Process," *New Paradigms for Software Development,* IEEE Computer Society Press, 2-5 (1986)

[5] Blum B. I. "The Life Cycle - A Debate Over Alternate Models," *ACM Sigsoft,* 7(4), 18-20 (1982)

[6] McCracken D. D., Jackson M. A. "A Minority Dissenting Position", *Systems Analysis and Design-A Foundation for the 1980's,* edited by W. W. Cotterman et al., 1981, 551-553 (1981)

[7] Curtis B., Krasner H., Shen V., Iscoe N. "On Building Software Process Models Under the Lamppost," *Communications of the ACM,*

A METHODOLOGY FOR SOFTWARE MAINTENANCE

by
John E. Moore

Hercules Incorporated
Bacchus Works
Mail Stop N1EB4, Box 98
Magna, UT 84044
(801) 251-6191

ABSTRACT

This paper describes a methodology which evolved over a period of years for a scientific software (mostly Fortran) maintenance environment. In this methodology, software maintenance is managed as a series of separate projects or releases to existing systems. The methodology consists of four phases: Requirments Definition, Implementation, User Acceptance, and Production. Testing and documentation are not separate phases, but are integral parts of requirements and implementation. The methodology is short and simple, which has contributed to its use.

Details of the methodology and its implementation are described. Although the methodology was originally based upon the classic waterfall or life-cycle model, it evolved to more closely resemble the evolutionary systems delivery model of Gilb. A discussion of attempts to measure software quality and the impact of the methodology are given. The paper closes with a summary and a list of DOs and DON'Ts from the author's perspective for anyone considering implementing a methodology for software maintenance. This work is experimental in nature and would be of most interest to managers who are contemplating methodologies to improve quality or productivity.

BIOGRAPHICAL SKETCH OF AUTHOR

John E. Moore is a supervisor of scientific programming for the Bacchus Works of Hercules Incorporated, which manufactures large solid propellant rocket motors. He holds a B. S. in Mathematics and a M. S. in Mechanical Engineering from the University of Utah. John has authored software engineering articles for the Software Maintenance News and the Fourth National Software Maintenance Conference.

A METHODOLOGY FOR SOFTWARE MAINTENANCE

by
John E. Moore

Hercules Incorporated
Bacchus Works
Magna, UT 84044

I. INTRODUCTION

Although software development methodologies have been popular for about 15 years, few have taken into consideration the special challenges of software maintenance. The classic waterfall or life-cycle model defines phases for Requirements Definition, Analysis, Design, Coding, and Test falling into a murky, ill-defined, and forgotten pool called Maintenance [Figure 1]. Nearly all of the waterfall models, and even most of the newer methodologies, such as rapid prototyping, assume most software is developed from scratch. Unfortunately, little attention is given to software that has been implemented, has developed a wide user base, and needs to be changed.

Methodologies are typically introduced to solve a software "crisis" which manifests itself in the perception that software projects are done inefficiently and are of poor quality. There are multi-year backlogs, a high percentage of project failures, and large programmer resources devoted to software maintenance, instead of new development. Perhaps most methodologies avoid maintenance because it is something they are trying to eliminate. However, there are many companies and programming environments for which the majority of software work can be classified as maintenance.

Parikh and Zvegintzov [PARIKH-82, pa. 1] show that maintenance as a proportion of programming resources has remained constant at about 50% since 1969 for most computing organizations. In the Scientific Programming Group at Hercules, the figure is closer to 90%. Some authors, like Martin and McCLure [MARTIN-83], view maintenance only as a necessary evil until new technologies, like Fourth Generation Languages and ADA, can replace the existing software. But others, like Zvegintzov [PARIKH-82], take a more realistic view. Like it or not, any effort to improve quality and productivity which ignores software maintenance is neglecting a large part of the programming resources. Maintenance is most likely here to stay.

Some techniques have been developed for improving software maintenance, such as Warnier/Orr Structured Maintenance [PARIKH-81 and HIGGINS-81], but few guidelines exist for implementing these techniques as a managed procedure or methodology. How does a manager ensure that techniques are being used correctly and are really contributing to quality and productivity?

Perhaps a reason few methodologies deal with maintenance is that it is perceived to be a disorganized and unmanageable activity. Maintenance programming can be very time-consuming and error-prone. Many existing systems were developed before structured programming, modularity, and maintainability were popular ideas. These programs were written for computers with severe memory limitations. They can be very complex, unstructured, undocumented, and poorly designed. They often suffer from the effects of 15 years of quick fixes. A one line repair can take days, even weeks. It can take months before a new programmer can become knowledgeable enough to make any useful modifications. Phillips [PHILLIPS-88] claims average maintenance programmer productivity to be one or two changed lines of code per man-day compared with 10 to 15 lines of code per man-day for new development.

Many of these systems are supported by only one programmer, who cannot be replaced. In

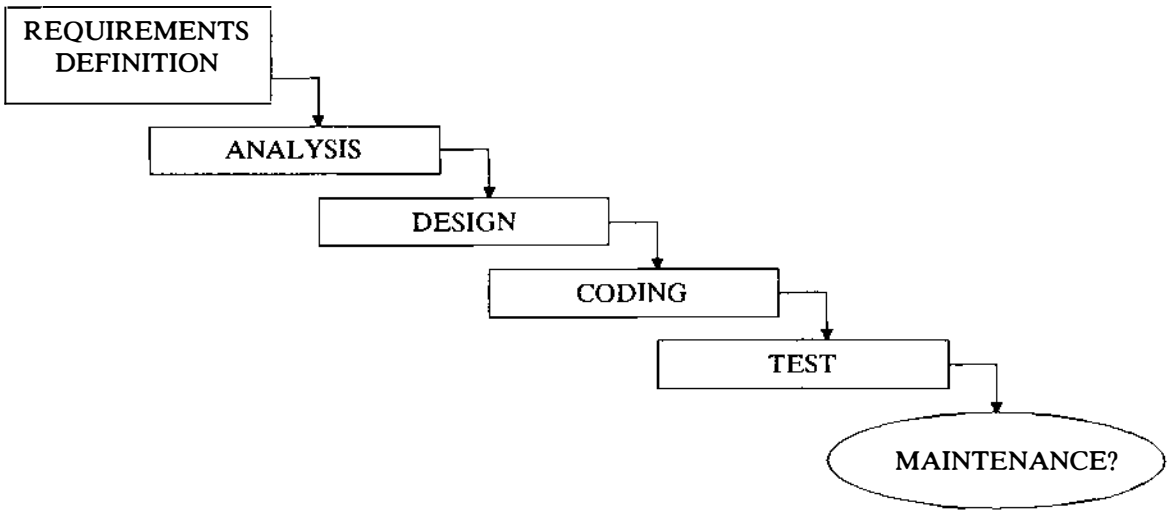


Figure 1. The traditional life-cycle or waterfall model for systems development.

one survey, 63% of the respondents indicated they had certain programs or systems that had to be maintained by "certain" people because no one else understood the logic [SMN-87].

A significant problem for anyone trying to organize the maintenance process can be the people. Sometimes customers and "irreplaceable" programmers have developed ingrained attitudes about what to expect from one another. A programmer can become very close to one set of customers so that work which comes from them is more likely to get done. The "irreplaceable" programmers are sometimes quite satisfied with their performance, and are not interested in changing their habits, which have obviously been somewhat successful, or the same software wouldn't still be around after 15 years.

The traditional craft approach to computer programming, which various software engineering methodologies try to supplant, is very evident in software maintenance. Implementation will be used to define the design of a change, and user testing to define the requirements. This approach might be called "non-rapid prototyping," or "code first, ask questions later." Programmers with years of familiarity with a particular program and its customers can sometimes make this approach work. However, the long term impacts on program maintainability are questionable. Less experienced maintenance programmers are often far less successful with this approach. As many authors have shown, defects are far more costly to correct after code has been written or changed, than during the early stages of a project. This craft approach is a prime contributor to quality and productivity problems.

Despite the negative aspects of software maintenance, it is still often more cost effective to add new capabilities to an existing program rather than to write a new one from scratch. New development has many of its own problems.

The rest of this paper describes efforts to solve scientific and engineering software maintenance problems at Hercules through methodologies. A methodology is a set of principles, practices, and procedures for doing a task. For the purpose of this paper, a methodology will be defined as formal principles, practices, and procedures which are managed, in contrast with informal practices which are encouraged. Such a methodology has control points or phases which allow an independent, but knowledgeable, reviewer to verify that tasks are being done according to procedure. Methodologies appeal to management because once a procedure demonstrates itself successful on a particular project, the same procedure can be written down, codified, and used on similar projects, hopefully, with the same successful results.

Section II, BACKGROUND, discusses early attempts to implement a traditional life-cycle methodology. Section III, MAINTENANCE

METHODOLOGY gives details of the methodology which evolved over four years. Section IV, EVALUATING SUCCESS, discusses attempts to measure the impact of the methodology. Section V is a summary, and Section VI provides advice for others trying to implement a software maintenance methodology.

II. BACKGROUND

The Scientific Programming group at the Bacchus works of Hercules Incorporated consists of approximately 20 programmer/analysts who supply engineering analysis software for rocket motor design engineers. Specific applications are in finite element structural analysis, ballistics, strength-of-materials, and composite materials design. Although some commercial engineering analysis packages are now available, the nature of Hercules products and materials necessitates the use of in-house developed custom software. Many of the important engineering analysis programs were originally written over 15 years ago in Fortran IV. Some are still state-of-the-art in their capabilities. These programs are very technical and very complex. Most programmers have an applied math or engineering degree with emphasis in numerical analysis. Successful programmers have a good understanding of both engineering theory and computer programming.

The Hercules Bacchus Works is in the business of making rocket motors for strategic defense and space applications. Hercules strives to provide a quality, affordable product (rocket motor) to its defense and space customers. Hercules does not sell software. Engineering analysis software is a critical design tool which enables Hercules to develop very complex, state of the art products and gives it advantage over its competitors. Programmers are under considerable pressure to modify or repair on short notice computer programs used by engineers to make critical design decisions.

The Scientific Programming group experienced many of the quality and productivity problems cited previously. There were projects which weren't completed successfully, much of the user documentation was out of date or incorrect, new programmers found it difficult to become productive, and complaints from customers were getting louder and more frequent. The problems were exacerbated by a large influx of new programmers and engineers who were hired in a short period of time.

The first effort to improve software development and maintenance was standards to enforce the basic concepts of structured programming. These standards were based on military standards [NAVY-78] and other sources from the late 70's and early 80's.

Although the standards introduced the concepts of modern programming, there was still a basic perception that software tasks were not getting done efficiently and were behind schedule. Two salient facts stood out. First, the group did very little new programming, since most of the work was modifying existing programs. It was very difficult to apply structured programming standards to 15 year old Fortran. Second, ensuring a program was structured did not have a big effect on expensive mistakes made early in the requirements or design.

This prompted development of a life-cycle methodology based on the popular software engineering articles of the early 80's. The life-cycle methodology was sold to management and adopted. A two-week software engineering class on the basics of structured analysis and design was taught on-site by a consulting firm to all programmer/analysts. A new standard, called the System Development Standard (SDS), was written by programmers. It was based on the guidelines of Victor Weinberg [WEINBERG-80]. This methodology had eight phases:

- Problem Statement
- Requirements Definition
- Feasibility Study
- Detailed Logical Design
- Implementation Plan
- Detailed Physical Design
- Implementation and Testing
- Project Evaluation

Each phase had a defined output. Walkthroughs were to take place after each phase to ensure that all projects were being done correctly. A big improvement in performance with all these standards and training should have been evident. Unfortunately, there were still problems.

Initially, most programmers did not use the standards because they did not know how to apply them to their work, which was often modifying an existing program. To address this dilemma, software maintenance was organized into releases

[KONTOROVICH-85]. Enhancements and corrections to a program were bundled together to form a new release of a program. It was intended that the same SDS phases would apply to each new release.

Theoretically, the customers should have been overjoyed to see all the organization and standards. However, it took only a few programmers saying, "I'll get to your project as soon as I finish this SDS requirements definition," for the standards to be perceived as bureaucratic and unnecessary paperwork. Many engineering customers were used to just going to a programmer and asking for a change to a program. Whether the changes were correct or not was another matter. The SDS created a new level of formal communication and review which some engineers and programmers did not appreciate.

Although the standards had a positive influence on some programming efforts, expected improvements in quality and productivity, and reduction of customer complaints were not realized. Many programmers made good faith efforts to use the new standards. Some projects were completed successfully, but others which also used the methodology weren't. Many of the problems associated with "non-rapid prototyping" were still evident. A number of problems stood out:

(1) Although the programmers had two weeks of structured analysis and design training, many were unsure how to apply the concepts to their work. Most programmers did not have a software engineering educational background and the concepts were very new to them. Most universities teach very little software engineering to their students, even those with computer science degrees. There were very few mentors available for help.

(2) Most software maintenance projects, even those requiring significant program modifications, were relatively small projects. The nine phases of the SDS seemed like excessive bureaucracy to some programmers and engineers. Even though the SDS allowed fewer phases to be done for small projects, programmers lacked the experience and judgement to effectively use the standards without getting bogged down in details.

(3) The logical and physical design phases were very difficult to apply to software maintenance. How do you do a logical design for a change to a program which has no design? Is it worth the effort? How do you walk through the design?

(4) The SDS was an abrupt culture change for some programmers and customers used to an informal system.

(5) Despite the guidelines given by Yourdon [YOURDON-78], walkthroughs did not work well in providing quality control. Unlike artists and architects, which have their work subjected to peer review in school, scientific programmers have no educational experience in having their work critiqued independently. They were uncomfortable with it and avoided doing it voluntarily.

(6) To be effective, walkthroughs require the reviewers to be independent, but knowledgeable. This was very difficult in our engineering environment. Often it took months for a programmer to competently understand a program and its application. This made it difficult for other programmers to review design and programming decisions without knowing a great deal about the application and the program. The programming group is small with many varied and complex applications to support.

(7) The Requirements Definition phase of the SDS was not working well. There were too many projects that failed even though requirements were written. Programmers are not notoriously good writers, and many customers were merely glancing at the Requirements Definition documents, and signing them, so the programmers could get on with their "real" work.

(8) The standards were difficult to enforce and manage. Despite the sign-off documents produced, it was difficult for management to determine the condition of a project before it was too late. The walkthrough guidelines state that management should not attend walkthroughs, which meant management had to rely on forms signed by programmers who were not always willing to have their work reviewed.

(9) Documentation was saved for last and did not always get completed.

Although our experience cannot be perceived as a blanket condemnation of the life-cycle methodology, our implementation did not work as well as expected (a more virulent critique of

the life-cycle methodology can be found in AGRESTI-87). There were many positive benefits which resulted, especially in improved attitudes towards design, requirements and quality among the programmers. However, it was obvious there was still room for improvement, and the methodology would become nothing more than a shelf potato unless something was done.

III. MAINTENANCE METHODOLOGY

Since our implementation of the traditional life-cycle methodology wasn't working as well as we hoped, we reviewed our successes and failures, and developed a new strategy:

(1) Concentrate on the most cost effective parts of the methodology for the software maintenance environment and turn them into effective tools that the programmers could understand and use. This resulted in concentrating our efforts into making the requirements definition phase work effectively for software maintenance.

(2) Formalize only those parts of the process which could practically be managed. Encourage correct informal practices, but don't formalize them until they are well understood and manageable. This resulted in our not requiring procedures which had marginal benefit to software maintenance, like logical and physical design.

(3) Try to package software projects into about three or four man-months of work. Our biggest failures had occurred on projects which had not delivered a useful product to the customer in six or more man-months of work. It is relatively easy to organize software maintenance tasks into smaller projects. Small projects cannot get too far out of hand without the results becoming evident to programmers, customers, and management.

(4) Try to make an impact on the future maintainability of programs. Since a large portion of a maintenance programmer's effort is in studying source code and documentation to understand how a program works, improvements to programmer documentation provide one of the single largest improvements in future maintainability. Review of source code comments and documentation is also easy to implement via formal rules and procedures.

This approach resulted in a methodology which more closely resembled the evolutionary systems delivery model of Gilb [GILB-87] than the traditional life-cycle. The maintenance methodology is customized to the software maintenance environment and people. Marginally cost effective procedures are not required. Projects are kept small. Effort is concentrated on procedures with the highest short-term value to software maintenance.

A. Project Management

The importance of organizing software maintenance into release packages cannot be underestimated. A beginning and an end is defined for each project. It has a schedule and an estimated budget. The traditional non-rapid prototyping approach often results in hiding the source of failures and inefficiencies from view. Conversely, small man-month projects and a few well defined phases, make projects visible enough for control by management and customers.

Software maintenance is made up of corrections, enhancements, adaptations, and perfections. Corrections address errors and omissions. Enhancements add or delete capability. Adaptations allow use in new hardware or software environments. Perfections are program improvements which enhance future maintainability. By bundling all of these together into a project, all modifications to a program can be tested together to reduce the chance that one change will negatively influence another. Each project goes through four phases:

- Requirements Definition
- Implementation
- User Acceptance Testing
- Production

Requirements Definition is not approached in the abstract way typical of many life-cycle methodologies. Rather, this phase tries to get as much information from the customer as early as possible, with feedback of the requirements to the customer in a form he will understand. Documentation requirements and test cases are specified early in the projects, and are not saved for later.

Implementation represents the actual code and documentation changes and their verification. Because of the varied nature of engineering programs and their documentation, formalizing this

process is difficult. Practices such as structured programming, re-engineering, and modularization are encouraged, but not formally regulated. Instead, the formal procedures concentrate on ensuring the programmer documentation is complete, meets standards, and has been subjected to peer review.

Customer acceptance testing is not a substitute for programmer testing. Rather, it provides an opportunity for the customers to verify that they are getting what they paid for.

Production is the formal process of moving source code, load module, and documentation into protected libraries. Once a new release goes into production, the project is completed.

The schedule and budget for a new release to a program need enough flexibility to accommodate error correction. Errors can be identified at any time in a production program. They can be classified as either annoyances or catastrophes. Annoyances are problems needing correction, but some type of work-around exists for the customer so he can wait for a new release with the correction. Catastrophes, on the other hand, require immediate attention for the customer to do his work.

Work Request # 77118
Program # 75787, 75851

CADAM GRID GENERATOR SCHEDULE

Date 21 June 1988

TASK/WORK ELEMENT	MAY	JUNE	JULY	AUG	MAN DAYS (MD) USER	MAN DAYS (MD) PROGMR
RELEASE 88.2						
• Change CADAM file naming to HOLD						
• Set precision of CADAM-to-GRIDGEN data						
• Correct TSEC problems going to CADAM						
• Send HICAP vectors to CADAM						
• Interface splines and conics option						
• Incorporated new keyword cards option						
Requirements Definition						35
Implementation						245
• Documentation						35
• Testing						35
User Acceptance					35	
Production						1
				TOTALS	35	351
USER (MD)	0	0	35	0		
PROGMR (MD)	192	158	1	0		

Figure 2. Typical Software Release Project Management

Catastrophic errors are corrected as soon as they are identified. A temporary test version of the program with the corrections is created which allows customers to get on with their business. However, all error corrections are tested along with other changes before a new release goes into production. In a new release, all modifications are tested together.

A typical budget and schedule chart for a maintenance project is shown in Figure 2.

B. Requirements Definition

The importance of good requirements for a software maintenance project cannot be over emphasized. Although experienced, irreplaceable programmers may be successful with informal techniques, new maintenance programmers can benefit greatly from good maintenance requirements.

The requirements definition is a tool for communication between customer and programmer. It is also a tool for helping the customer clarify to himself what he wants. It should therefore be written in the customer's language. Each sentence or picture should generate some sort of response, either positive or negative, from the customer. If the requirements are too abstract or nebulous, the customer will very likely gloss over them, and not make meaningful contributions. He will just sign the requirements so the programmer can "get on with his work."

Since maintenance projects deal with systems which already exist, reference to the existing physical system through examples and marked up copies of the input or output interfaces (such as input panels, input files, and output listings) is essential and natural. The customer often cannot relate to abstract concepts such as accuracy and adaptability. But he might be able to relate to an important table in the output listing. In some ways, the requirements definition can be thought of as "paper prototyping."

One example of the importance of "paper prototyping" occurred when requirements had been written for a change to a program in the traditional abstract way. The customer signed the requirements hoping the project would get started. When the requirements were rewritten to include penciled in copies of the input and output which showed the impact of the maintenance change, the customer suddenly took much more interest and caught a number of significant omissions in the requirements which weren't evident to him before.

Figure 3 shows the format and organization of the requirements document. These requirements are not a legal document. They do not need to be typed, but must be in a form which communicates. They need to be approved and signed by the customer and programmer before they are completed.

Note that test requirements and documentation changes are specified at the beginning of the project and are not reserved for a later phase.

C. Implementation

Only two formal requirements are made of the implementation: (1) that the source code comments and other programmer documentation be subjected to peer review via walkthrough, and (2) that the user documentation be reviewed and accepted by the customer. These minimal requirements on implementation contrast strongly with the logical design, physical design, code walkthroughs and other requirements of life-cycle methodologies.

About 30% of the average maintenance programmers time is spent looking at the source code and documentation to understand how a program works [PARIKH-82, pa. 2]. Accurate and clear programmer documentation is the largest contributor to maintenance programmer productivity. This documentation, in the form of source code comments and programmer manuals, can be easily subjected to peer review. Standards for subroutine and program header comments provide requirements and example formats for the programmers to follow. Other programmers have a vested interest in reviewing comments which they might someday have to read, and in many cases it does not require an extensive knowledge of the application.

In some cases, a logical or physical design can be made of a maintenance change using data-flow diagrams, structure charts, or Warnier-Orr charts. However this cannot be made a formal procedure due to the widely varying condition of existing programs and the nature of the changes requested. Some programs have undergone so many quick fixes over the years that there is not much

I. **STATEMENT OF THE PROBLEM**

A. PROBLEM STATEMENT
- what is the problem? 1 paragraph description of customer's perceived problem

B. HISTORICAL PERSPECTIVE
- where did the problem come from? 1 paragraph description of events leading to problem (optional)

C. PROBLEM IMPACT
- who is affected by it? List names and groups.
- what projects are affected? List names and Hercules customers (optional)
- what justifies solving the problem (cost, quality, or safety)?

II. **REQUIREMENTS**

A. FUNCTIONS THE ENHANCEMENTS MUST PERFORM
- 1 or more paragraphs, tables, drawings, or equations describing what the solution must do.
- what are execution or response time maximums or minimums? (optional)
- what existing options will the enhancement affect?

B. INPUT REQUIREMENTS
- what will the inputs look like? Provide a paper prototype of the input using a penciled-in copy of the existing input format as an example
- describe input data items, their type, range, volume, and accuracy.
- what error or warning messages are needed? Provide paper prototypes of the error messages.

C. OUTPUT REQUIREMENTS
- what will the outputs look like? Provide a paper prototype of the output using a penciled-in copy of the existing output as an example. Use tables, plots, files, etc. to show the customer in detail the expected outputs.
- describe output data items, their type, range, volume and accuracy.
- what error or warning messages are needed? Provide example output error or warning messages.

D. USER DOCUMENTATION REQUIREMENTS
- what documentation is required? Provide a copy of the existing documentation, or table of contents, and pencil in where changes will occur.
- who will review the documentation? Provide a review group list.

E. ACCEPTANCE TEST
- what will verify that the problem is solved correctly? Give input and corresponding output for all test cases.
- what is expected shape of plot curves?
- is there measured data, or another validation technique to which the outputs will be compared with?

III. **CONSTRAINTS**

A. HARDWARE
- i.e. IBM 3090, Apollo, IBM PC

B. LANGUAGE
- i.e. IBM VS FORTRAN, C, Turbo-Pascal, etc.

C. STANDARDS & PROCEDURES
- SPA Coding Standards, SPA System Development Standards.

D. SAFETY
none

E. COMPATIBILITY WITH EXISTING SYSTEMS
- what potential compatibility problems could occur with existing software or systems. (i.e. the Grid Generator output can't be changed without impacting 62114, etc.)

IV. **PRELIMINARY LOGICAL OR PHYSICAL DESIGN (optional)**

Provide DFD's, Structure Charts, and/or Warnier-Orr charts.

V. **PROJECT SIZE & TYPE**

Small modification project.

VI. **COST ESTIMATES FOR NEXT PHASES**

Attach project planning and scheduling charts

Figure 3. Requirements Definition

control or data structure left. Also, applying these concepts to software maintenance and making them work effectively requires significant software engineering skill, which most maintenance programmers do not have as yet. Requiring a logical or physical design in these cases is pointless. Peer review, through walkthroughs, is also very difficult and inefficient due to the complex nature of most applications. These same limitations apply to coding. There are too many different environments and applications to formalize the process, and make it work in practice.

However, analysis, design, structured programming, and other modern software engineering techniques are encouraged as informal practices. Code restructuring, re-engineering, modularization and reformatting are also encouraged. Tools are provided, and classes taught to support these techniques. Perhaps as the programmers gain experience, and if it becomes cost-effective, these practices might be added as formal requirements to the methodology in the future.

Requiring maintenance projects to be small supplants the need for multiple phases, such as logical and physical design. Every four man-months of effort or less should result in new capabilities delivered to the customer through a new program release. Projects which are doomed to failure will become obvious in a relatively short period of time.

The implementation phase is not considered complete until the program has passed all the test cases, and the customer documentation is complete. A regression test suite and automated testing tools are used for programs which have them. The customer documentation is reviewed and approved by the customers.

C. User Acceptance Testing

The primary purpose of this phase is to schedule and budget the customer resources needed for approving the maintenance release as complete. Acceptance testing is not a substitute for programmer testing, but it does allow the customer to test drive the software before it becomes production. Budgeting for the phase allows it to have more visibility as an important function for the customer to support.

D. Production

Production is the formal change control procedure for placing the source code, load module, and documentation in protected libraries. Formal standards and sign-off sheets exist. The primary purpose of Production as a separate phase is to provide visibility in the schedule and budget to the final step which will complete the project. A project is not completed until the new release actually goes into production.

IV. EVALUATING SUCCESS

Gilb suggests using a statistical process quality control model ala Demming or Durant to control the software development or maintenance process. Gilb describes a large number of data items for measuring quality attributes such as availability, maintainability, usability, etc. Data is collected and charted on items such as mean-time-between-failures, mean-time-to-repair and mean-time-to-improve. These objective measurements should then replace the subjective feelings that influence judgments about what's right or wrong with the process. Theoretically, the implementation of an improved methodology should result in improved software quality through a reduction in these non-conformance measurements.

An initial attempt is being made to measure the impact of the maintenance methodology on quality. The measurements are mean-runs-between-failure and the price-of-nonconformance (PONC). Unfortunately, these efforts began after the maintenance methodology was implemented, so no data is available on the previous process. Since the engineering software is run by the customers themselves, they determine how often it is run and for how long. The customers are the only ones who can realistically determine what constitutes failure in the software.

To collect data to determine the impact on quality, an on-line system was developed which allows customers to easily record failures and the associated costs they experience with most of the major software systems they use. This data can then be charted and prominently displayed. The system is very new, and some of the problems with data collection in support of statistical process quality control are evident. The present infrequent use of the system indicates lack of customer incentive to

record non-conformances. No credible data has been collected yet.

In theory an improvement in the software maintenance process via a new methodology, should result in a reduction in the number of failures and PONC. Unfortunately, there will be many other factors which can influence the data, such as informal practices, personnel turnover, etc. This points to the difficulty of measuring the effects on a process, and identifying the corresponding causes. The measurement effort will continue, but there are unanswered questions about its ultimate value and the difficulty and cost of collecting the data, especially for such a small programming organization.

The death of a methodology occurs when it sits on a shelf gathering dust. If the methodology is not being practiced, there is no way to determine its impact on quality or productivity. Despite the implications of many authors, we have found that a formal process (methodology) will not implement itself. Although some programmers willingly use new techniques to help them improve quality and productivity, many will not unless they perceive immediate and impressive results. If a programmer has a choice between writing a requirements document and not writing one, his natural tendency is to choose the path of least effort.

This points to an important quality attribute of methodologies which is not discussed by most authors. The methodology must be manageable. The manager must be able to tell, with a minimum of effort, whether or not the methodology is being used, who is using it, and how it is being used. A key component of the methodology described in this paper is that its simplicity contributes to its manageability.

Tom DeMarco quotes the axiom "you can't control what you can't measure." For the next few years, until advances are made in the theory and practice of software maintenance metrics, the primary measurement tool will still be abdominal tactile response, or gut feel.

V. SUMMARY

Our initial implementation of the traditional life-cycle model for software development proved not to work well for software maintenance. Improvements were made by formalizing procedures which had obvious benefit to software maintenance and not formally requiring those which were of marginal benefit. The simplicity of the methodology contributed to its acceptance and use. Initial attempts to measure the impact of the methodology have been inconclusive due to the difficulty of collecting the data. However, manageability of the methodology has emerged as an important attribute which cannot be ignored.

VI. DOs AND DON'Ts FOR IMPLEMENTING A METHODOLOGY

DOs:

- Start the methodology small and work up. Add on features which are proven and have the most benefit, are most cost effective, and are most manageable.

- Remove features from the methodology which don't work.

- Market the methodology as a quality and productivity enhancement tool. Never forget the original purpose of the methodology, which is to enhance quality and productivity through improved communication.

- Include rules and guidelines in the methodology. Separate them if necessary, but don't discard one or the other.

- Ensure that the overhead associated with implementing and using the methodology is perceived as being offset by the benefits of more successful projects, improved quality, and improved productivity. The methodology must be cost effective.

- Define "success" for the methodology before implementing it. Begin collecting quality and productivity data before implementing the methodology.

- Find some small projects with high probabilities of success and with people willing to try the methodology before codifying it. Demonstrate success early.

- Combine training with the implementation of the methodology. Be prepared for programmers and customers who won't know how to use the methodology correctly.

DON'Ts:

- Avoid implementing a large unwieldy procedure at the beginning. If the methodology is perceived as too big, too complex, or too cumbersome, it will be perceived as an impediment by both customers and programmers to doing "real" work.

- Don't expect a methodology to solve all your quality, productivity, and organizational ills. There will still be projects that fail no matter what methodology you use.

- Avoid labelling ancient, unstructured, undocumented, poorly designed software as "bad." It will just make the programmers who wrote it mad.

- Don't assume everyone will love the methodology. Its implementation will involve much more marketing and politics than technology.

- Don't allow the words "bureaucracy" or "paperwork" to be used in the same sentence as "methodology."

- Avoid codifying the methodology until you are prepared to manage it and willing to accept the resulting consequences and expenses.

- Don't give up.

VII. REFERENCES

AGRESTI-86 Agresti, W., New Paradigms for Software Development, IEEE Computer Society Press, 1986.

GILB-87 Gilb, T., "Advanced Practical Management Methods and Technologies for Controlling Software Maintenance Cost, Productivity, and Maintainability," course notes provided at the IEEE Conference on Software Maintenance-1987, September 21-24, 1987, Austin, TX.

HIGGINS-81 Higgins, D. A., "Structured Maintenance, New Tools for Old Problems," ComputerWorld Magazine, June 15, 1981. Also found in PARIKH-82 pa. 95.

KONTOROVICH-85 Kontorovich, F., "Implementing the Release Method," Third National EDP Software Maintenance Conference, Conference Notebook, May 6-8, 1985, Dallas, TX.

MARTIN-83 Martin, J. and McClure, C., Software Maintenance: the Problem and its Solutions, Prentice-Hall, Inc., 1983.

NAVY-83 Military Standard for Software Development, DOD-STD-1679A (Navy), U. S. Government Printing Office, Oct. 22, 1983.

PARIKH-81 Parikh, G., "Structured Maintenance the Warnier/Orr Way," ComputerWorld Magazine, Sept. 21, 1981. Also found in PARIKH-82, pa. 85.

PARIKH-82 Parikh, G. and Zvegintzov, N., Tutorial on Software Maintenance, IEEE Computer Society Press, 1982.

PHILLIPS-88 Phillips, R., "The Neglected Wasteland," ComputerWorld Extra Magazine - Productivity in MIS, June 20, 1988, pa. 42-43.

SMN-87 "The Maintenance/Month-Surveys," Software Maintenance News Magazine, March 1987, pa. 6.

WEINBERG-80 Weinberg, V., Structured Analysis, Yourdon Press, 1980.

YOURDON-78 Yourdon, E., Structured Walkthroughs, Yourdon Press, 1978.

Tools

"A Hardware Assistant for Software Testing"
Bill Sundermeier, MicroCASE, Inc.

"A User Interface Toolkit for the X Window System"
Benjamin Ellsworth, Hewlett-Packard Company

"Tina: A Facility for Assisting With Unit Testing"
Richard D. Sidwell and Gordon W. Blair, Rockwell International

A HARDWARE ASSISTANT FOR SOFTWARE TESTING

Bill Sundermeier
MicroCASE Inc.
19545 NW Von Neumann Drive
Beaverton, OR 97006

Abstract

The SoftAnalyst[tm] is a hardware monitor that applies coverage, path, and performance-analysis verification and testing techniques for improving software quality.

Biography

Bill Sundermeier has worked for three years at MicroCASE (formerly Northwest Instrument Systems). During the first two years, Bill was a design engineer for the group that developed the SoftAnalyst product. He has now moved into an Applications Engineering position to provide extensions to the SoftAnalyst product to apply it to software validation and verification.

Bill received a Bachelor of Science degree in Computer Science from Oregon State University. He is currently working on his MSCS degree at the Oregon Graduate Center.

Introduction

Software tool developers and instrumentation manufacturers have been neglecting the support of software testing and validation tools. There are numerous tools that aid in code generation and debug, but very few that assist in validating, testing, and improving quality. The SoftAnalyst, by MicroCASE, provides several different modes of operation that assist the software engineer in testing and validating programs by code and data coverage analysis, and performance analysis.

The SoftAnalyst is a hardware assistant that monitors the execution of software by connecting to the address, data, status, and clocking signals output by the target microprocessor. It does this in real-time without any intrusion or any instrumentation added to the code under test. It monitors code execution by relating symbol information generated by a compiler and linker to the absolute addresses in the target processor's memory where those symbols have been mapped.

The SoftAnalyst in the true sense is a dynamic testing tool which seeks to show the test engineer exactly what the program is doing as it executes. It is different than most other debugging tools, like emulators, that let the test engineer poke around in an ad-hoc fashion trying to find bugs in the program by setting breakpoints and examining registers and absolute memory contents. This is not to say that the SoftAnalyst can't be used for debugging, but to convey that the SoftAnalyst tries to enforce a more formal method of testing for validation.

The text describes, in the following order, the operational modes of the SoftAnalyst and how they relate to different software validation and testing techniques.

1.0 Program Coverage Analysis
- 1.1 Code Coverage
- 1.2 Data Coverage

2.0 Program Performance Analysis
- 2.1 Program Activity
- 2.2 Module Duration-Demand
- 2.3 Many-To-One Linkage

3.0 Path Analysis
- 3.1 Branch Testing
- 3.2 Specification Testing
- 3.3 Sequence Debugging

4.0 Virtual Trace

1.0 Program Coverage Analysis

The most common validation approach in today's software development life cycle seems to be ad hoc testing. In this approach, engineers test a program by running it through as many test cases as they think necessary. If bugs are found, they are fixed and testing continues. Finally, when the engineers have somehow tested the program enough (usually based on the number of hours spent poking around), they deem it verified. Although very simple, this method is quite unreliable in finding program errors.

Coverage techniques can be used to turn ad hoc testing into a more formal testing method. Coverage monitors give feed-back to the test engineer based upon what parts of a program have, and have not, actually been executed. The test engineer can turn ad hoc techniques into testing that insures all statements in the program have been executed. This usually covers the major functional tests that ad hoc would have, and also forces the test engineer to try cases that wouldn't have normally been thought of (the cases that the first users try). If a software group is using an ad hoc approach to verification, without some form of coverage analyzer, it is very likely that the code being put into production hasn't had every line tested.

Coverage testing can be designed into software so that it is instrumented with probes that write out information about where the program is executing. This requires overhead in both code and execution, as a probe must be inserted at every statement, or at a minimum in every sequential section of code. The amount of overhead may be acceptable in some situations, but not for real-time system environments, because it can actually change the program's behavior.

1.1 Code Coverage

CodeMap^{tm}, one of the SoftAnalyst's modes, provides coverage information by using hardware to monitor addresses accessed by the target processor. It accomplishes this without any modifications to the code, any recompilation, or any degradation in execution performance. A large memory array is created in the hardware monitor where each bit directly maps to a specific address in the target processor's memory. As the target processor executes and generates addresses, bits in the memory array are set for each address that is seen on the target processor's bus. The address information is then related to the symbol information put out by the compiler and linker to provide routine, variable, and line number coverage information to the user.

CodeMap provides statement coverage which shows the user exactly what parts of the program, line by line, have and have not been executed. It takes the address information from the bit array in the monitor's hardware and figures out which modules, procedures, and line numbers have been executed. Figure 1 is a picture of a CodeMap display. The left most column in the display shows the absolute addresses that were actually accessed. The middle column shows the symbol information for the part of the program at those addresses. It is titled by "Program Symbol," and shows the module's name, followed by the routine's name and line number within the module. The far right hand column shows the size, in bytes, of the code that was covered.

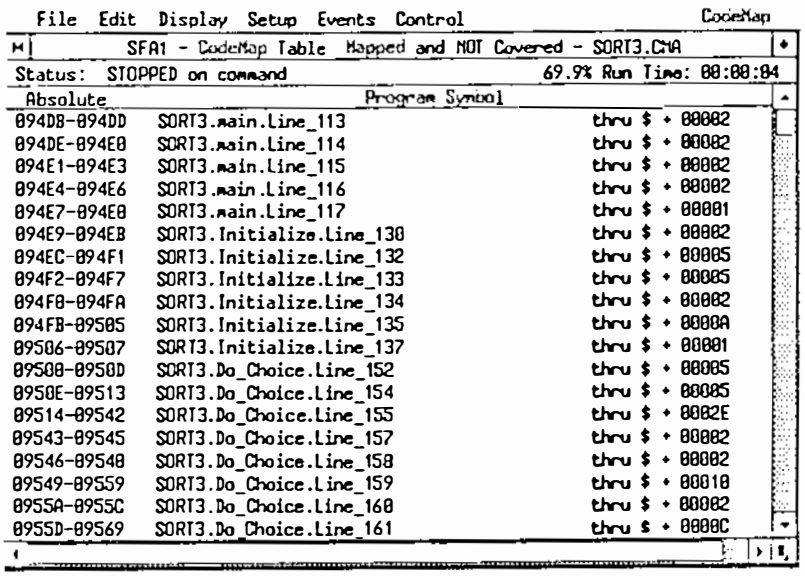


Figure 1. CodeMap shows what parts of a program have been covered by a test. This feature of the SoftAnalyst provides statement coverage, predicate coverage, and data coverage.

The display can be changed in a number of ways. It can show the code that was not covered, and also merge its data with other acquisitions to show shared, and not shared, covered areas.

CodeMap, from the program's symbol information, knows the entire size of the program and can show the percentage of the code that was covered in the measurement. On the third line of the display above "69.9%" is the percentage of the total program that was covered.

One of the most important advantages of directly mapping each address to the program is that it can show partially evaluated statements. This kind of coverage goes beyond simple statement coverage and provides a means to cover every instruction to the processor. This feature is used, for example, to insure predicate coverage in conditional statement evaluation. In languages such as C, compilers optimize code such that expressions, if possible, don't have to be completely evaluated. In a statement like:

```
if (expression_A || expression_B) {/* statement body */}
```

If expression_A is true, the body of the statement is executed without testing expression_B. To completely cover this statement it is important that expression_B is also tested.

Another feature of coverage testing is that it can assist the test engineer in finding more test cases for further regression tests. As the code is modified, the current test case set can be used to verify the functionality of pre-existing code. The new test data that is

used to cover the code that was untouched by old test sets will be added to make a new, complete, test set.

1.2 Data Coverage

Data coverage testing shows what variables, structure, etc. are accessed by a program. The object of this kind of testing is to make sure that all of the programs variables have been used. CodeMap can distinguish between code execution cycles, data cycles (read & write), and all other cycles that can be seen on the processor's status lines. If a test engineer wants to measure just program statement coverage, all they need to specify is code execution cycles. By selecting just read and write cycles, data coverage can show whether all variables have been initialized and used. This is of particular importance in finding anomalies such as when a variable is referenced, but never initialized. One pass over the program will show values that were read from, and another pass over the program, with the same stimulus, will show what values were written to. Variable accesses that are in error are those that were read from but not written to. Using this same technique, data writes can be monitored to show how a program's stack and heap are growing to insure that they are not overflowing their boundaries.

Examining reads and writes to memory doesn't have to be limited to data. How many times have you wondered whether your program was crashing because it was being corrupted, maybe by a stray pointer? By running the program until it crashes, CodeMap will show in the map whether there were any writes into the code space that may have destroyed the program.

2.0 Program Performance Analysis

Performance Analysis is one of the most critical validation phases for real-time software test engineers. Performance Analysis allows them to prove that the software under test meets its timing requirements. It is also used to show the engineer how to increase the performance of program by determining which routines are taking the most amount of time. Performance Analysis, in the SoftAnalyst, is accomplished in hardware by recognizing addresses that correspond to the program's symbols and storing them with a time value. The user selects symbols for the measurement and programs them as events for the hardware to recognize. The instrument, as it recognizes events during execution, assigns the time values to the appropriate symbols. For example, if the event that was recognized is a line number, the time value that was given to it would be transferred to the routine that contains that line number. The idea is that the granularity which most measurements require is routine performance, not how long it takes to execute a single source line.

The SoftAnalyst has three modes of operation that provide performance analysis: Program Activity, Module Duration-Demand, and Many-To-One Linkage.

2.1 Program Activity

Program Activity (figure 2) is the first mode, it provides occurrence counts for each event, and minimum, maximum, total, and average execution timing for routines. These values can be directly compared against the specification to prove correct operation.

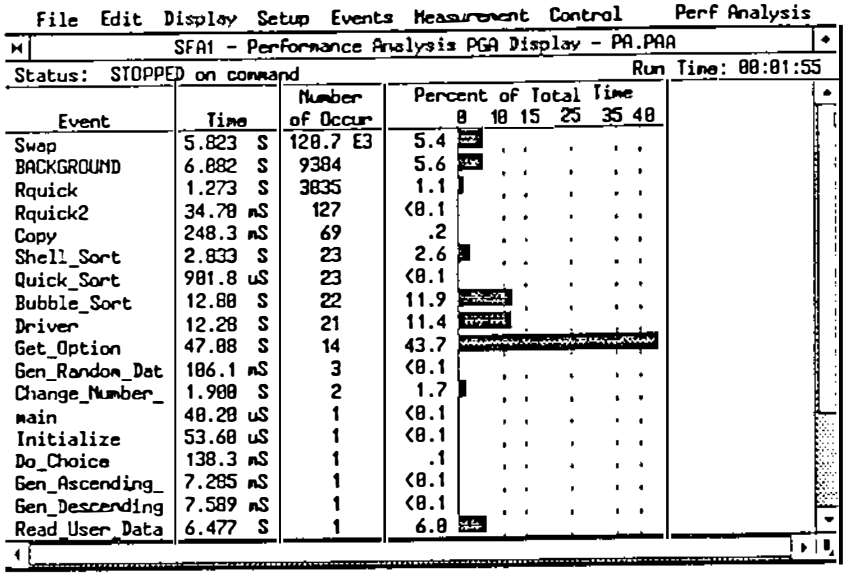


Figure 2. Program Activity is used to verify that a program is executing within its timing specification. It shows the total number of occurrences for each of the programs symbols, and shows timing information for routines and modules.

2.2 Module Duration-Demand

The second mode is Module Duration-Demand which provides a more detailed timing analysis for a particular routine. Module Duration-Demand, show in figure 3, allows the user to define several time intervals into which timing statistics are gathered. In this way, the user can get a "standard deviation" of the time that it takes for a routine to execute. Beyond min-max characteristics provided in Program Activity, Module Duration-Demand can show if a routine is erratic, consistently close, or far away from violating its timing specification.

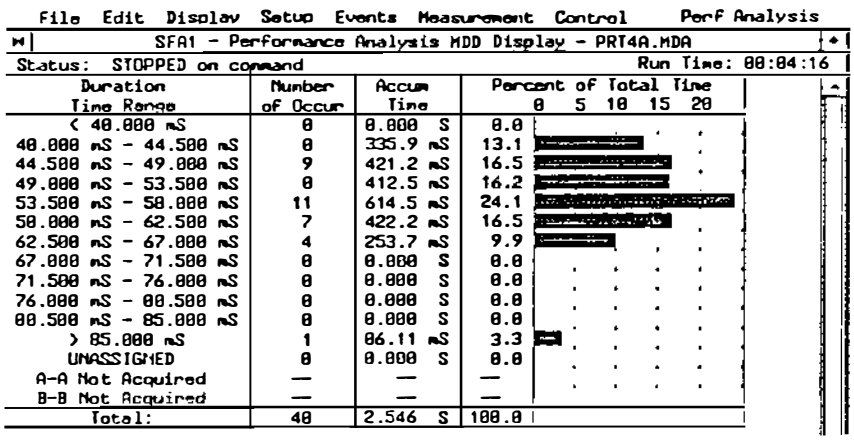


Figure 3. Module Duration-Demand is a performance analysis mode that shows the different amounts of time a routine takes to execute.

2.3 Many-To-One Linkage

The last mode of Performance Analysis operation is Many-To-One Linkage (figure 4). In this mode a single event is chosen and interactions with the rest of the routines in the program are measured. The most useful operations of Many-To-One are to determine who is using a routine, how often, and how much time the routine is consuming out of the total execution time. The mode is used for increasing the performance of the program, but the user can also quickly verify who is accessing a routine or variable and if there are routines that shouldn't be accessing it.

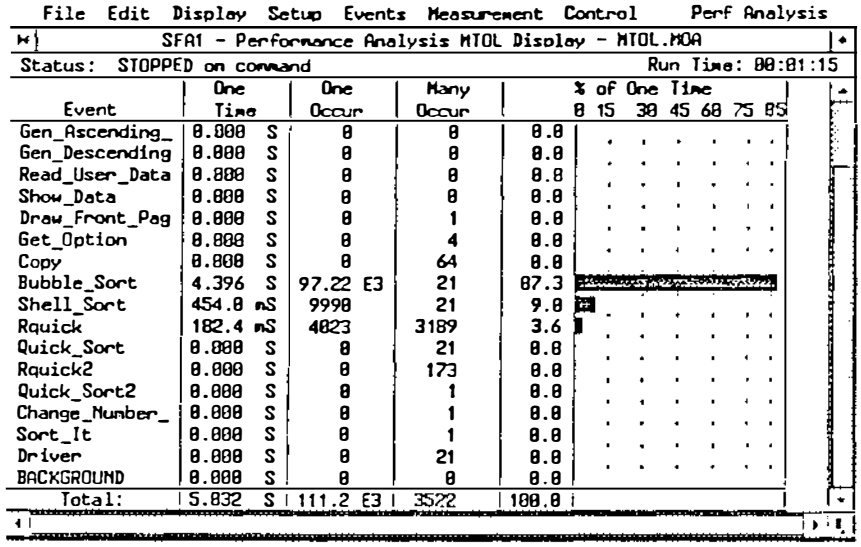


Figure 4. Many-to-One Linkage is a mode that shows the interactions between a single routine and other routines in the program. In this picture the one routine is "Swap", a routine that moves elements in an array for sorting purposes.

3.0 Path Analysis

Path Analysis is a method that examines the functional interactions of the elements of a program. It tests the code by monitoring how the program uses its routines, statements, and data.

SymTrace[tm] is a tool that is quite similar to a state analysis trace, but instead of seeing microprocessor bus level activity, the test engineer sees a high level trace of symbols (names of routines, variables, etc.) that are within their program. This type of trace can easily be related back to the program for determining execution flow.

3.1 Branch Testing

Branch testing is one of the most common forms of path analysis. It is done by testing each branch so that it has been executed at least once where it is taken, and once where it

wasn't taken. This method at least tests the branch paths thoroughly, and tests the code quite extensively because it has to be put through many cases to test all the branches.

SymTrace, in today's product, allows the user to capture a 700 to 900 event trace of their program's symbols. This is normally enough information to dynamically capture specific sections of the code under test. Test engineers can set up measurements to capture sequences until all possible branches have been executed.

In the following picture (figure 5) SymTrace shows the sequence of statements executed in one pass of a routine. If this case it has shown that a branch was taken such that line 8 was executed more than once.

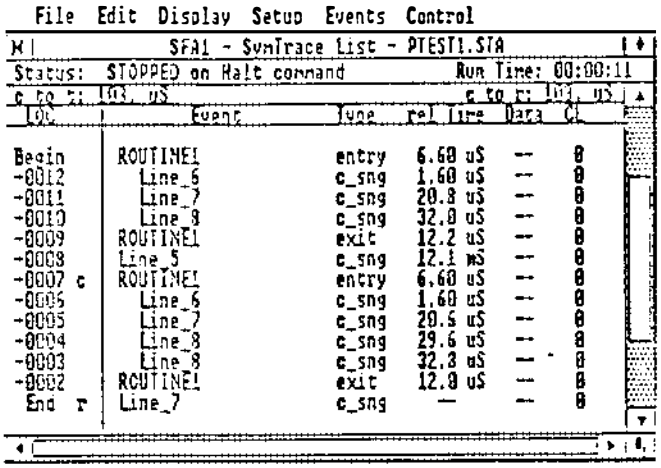


Figure 5. SymTrace shows the sequential flow of a program during execution. It is used to provide branch coverage information. In this case "Routine 1" was executed once where the loop at line 8 was taken, and once where it wasn't.

3.2 Specification Testing

Another interesting concept is the capability of verifying a program's code and data flow against its design specification. Module, function, and variable information can be extracted from the trace to prove that the code executed the proper sequences to perform a function described in the specification. Information such as who calls who, and who updated a variable can directly be checked against the specification. In the following figure (6) it is easy to see how the symbolic output of SymTrace can be used to check functional execution.

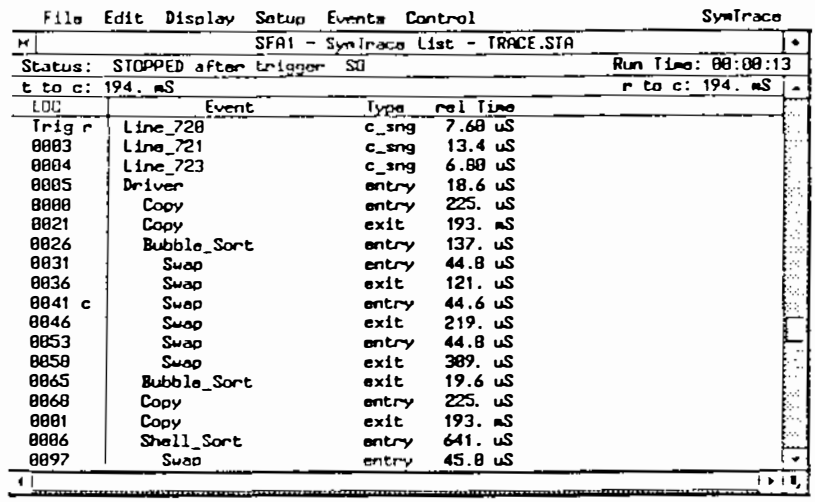


Figure 6. SymTrace can also be used to check the sequential flow of the program against specifications. The information can be used to verify who called who, and what values variables took on during certain routines.

3.3 Sequence Debugging

Path analysis is used primarily at the end of project to provide a better coverage metric, but examining execution flow can be used to find how and where problems exist in a program. Symbolic tracing can be used at the beginning of a project, for example, to determine where a program is crashing. To do this all that is required is to create symbols from the compiler/linker output for your program, let the instrument run, let the target run, and when the target program has crashed, stop the trace and examine where the program was before it crashed. The trace will show the last point executed before it crashed and the context in which the crash happened. Normally to find the same information an engineer would insert breakpoints or debug statements to determine where the crash happened. If debug statements are used, recompilation is necessary every time a new statement is added. Trace data acquired with hardware can be measured in a non-intrusive single pass over the program which eliminates the manual insertion of debug probes into the target code.

4.0 Virtual Trace

To increase the number of events which can be captured, Virtual SymTrace logs all acquired data to a disk on the host computer for later post analysis by user developed programs. Acquisitions from a program, even examining high-level symbols, can fill up 700 events and still not give the test engineer a complete trace of the program. With a virtual trace capability high-level program events can be measured from program start to finish.

Virtual SymTrace opens the SoftAnalyst to be used by customers to define their own measurements. The output of this trace is a published binary file that can be analyzed by

special programs. The output can be directly compared against other on-line outputs. One idea for this trace is that it could be compared against the source code to automatically determine what has been covered in terms of statements, branches, and paths. The output could also be compared against an on-line specification to validate the program against it.

The first post analysis program that was implemented measured routine time versus wall clock time. It shows the duration of how long a routine lasts and how often it is occurring. It was used to verify when an interrupt was happening and how long it was taking. In figure 7 the measurement assured the user that the interrupt being traced was happening with the frequency expected and that it was being serviced appropriately.

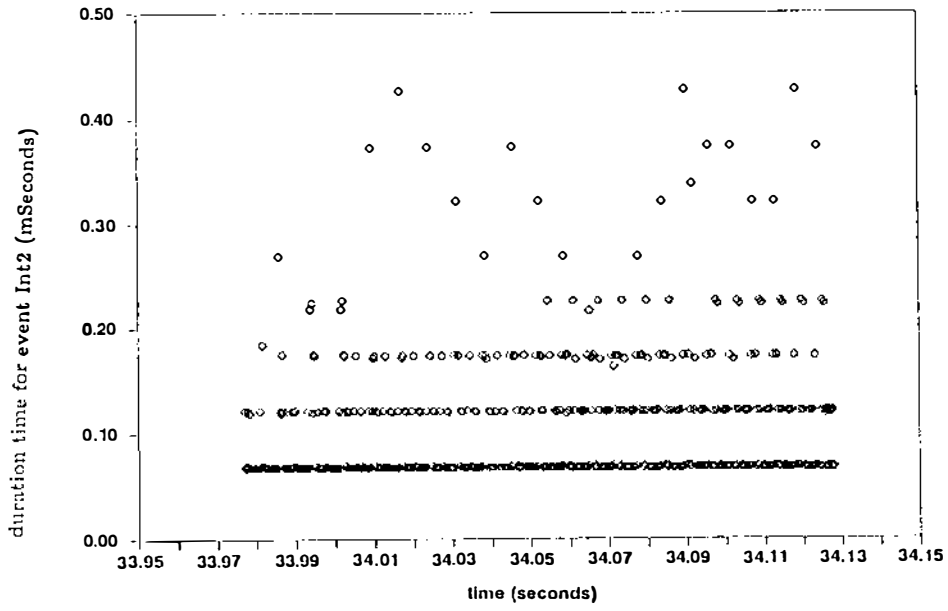


Figure 7. Virtual SymTrace is an indefinite trace of a program's symbols. It allows accessibility to the trace by user defined post-analysis programs. The picture above shows a post-analysis program that plots a routine's duration versus elapsed time.

Conclusion

There is a growing concern for the quality of software. As software controls more potentially life threatening machines and processes, the requirement of complete verification and validation becomes extremely important. Computers are being used in transportation, medical equipment, and other human interactive devices. How is the software being tested? Will the next bug in one of these devices cost someone their life?

The SoftAnalyst is a practical tool that can assist a test engineer in testing and validating critical aspects of their program. The SoftAnalyst is a product that has been on the market for about two years. Further work is currently being done with the product to increase its value in assisting with testing, validation and verification. There are many more exciting ideas to implement, from new measurement modes, to automating more of the test and specification checking techniques. Continued progress with this instrument, and other forms of verification and validation techniques, will help us to provide better tools for software developers to use and to improve the quality of their software.

A User Interface Toolkit for the X Window System†

Benjamin Ellsworth

Hewlett-Packard Company

ABSTRACT

When developing software that has an intuitive, graphical user interface, software quality and software engineering productivity can be significantly improved by utilizing a **User Interface Toolkit.** A toolkit consists of reusable **software objects** and general routines for their creation, manipulation and destruction. Example software objects are buttons, scroll-bars and menus.

This paper describes the development of a toolkit for use under the X Window System™ consisting of the **Xtk Intrinsics** and the **Hewlett-Packard X Widgets.** This paper also describes the experience of some current users of the toolkit and widgets in terms of productivity and quality.

ABOUT THE AUTHOR

Benjamin Ellsworth received his Bachelor of Science in Computer Science and Statistics magna cum laude from Brigham Young University. He has worked with Dr. Dan R. Olsen Jr. on the MIKE user interface management system. He immediately joined HP upon graduation working as a research and development engineer on interactive computer aided electronic engineering products with primary responsibilities in the area of gate array and standard cell design and verification. In 1987 he joined the toolkit development effort at HP, and is responsible for the design and development of a number of widgets and the separation of widget task from widget view. He may be reached at:

Hewlett-Packard Company
1000 N.E. Circle Blvd.
Corvallis, Oregon 97330-4239

ben%hpcvlx@hplabs.hp.com or hplabs!hpcvlx!ben

---

† "X Window System" is a trademark of Massachusetts Institute of Technology.

A User Interface Toolkit for the X Window System†

Benjamin Ellsworth

Hewlett-Packard Company

1. The Context

The X Window System™ is an emerging standard for network transparent window oriented graphics. Input and output is handled within a client-server model. A display server distributes input to and accepts output requests from client programs. Interface libraries for several programming languages have been implemented. The interface underlying all work discussed in this paper is **Xlib**, the interface library for the C programming language [1]. Protocol is defined for window creation and initialization, basic line drawing, basic polygon operations, basic text output operations, and generalized input events from a keyboard and a locator device.

Xlib does not include any high level definitions for any user interface paradigms; none are standard to the definition or specification of the X Window System™. The creation of common user interface objects is cumbersome at the Xlib level. It is much like user interface development within one of the common graphics standards (e.g. GKS): cumbersome and entirely unconstrained. An application which wishes to create user interface objects at the Xlib level must define and produce all graphical appearance, and support the drawing and redrawing of that appearance. Additionally and perhaps most painfully, that application must provide for all of its own input processing.

Because each program written at the Xlib level is responsible for its own implementation of its user interface at such a low level, code re-use is not well supported. The cost of re-implementing user interfaces on an application by application basis can be quite high. Figure 1 is a graph of two software metrics across several functional modules for a Hewlett-Packard software project prior to release [2]. In this figure the high cost of complete user interface implementation and the low cost of reused code can be seen. Cost can be measured both in the number of defects found and fixed prior to release, and the complexity which was designed and implemented.

---

† "X Window System" is a trademark of Massachusetts Institute of Technology.

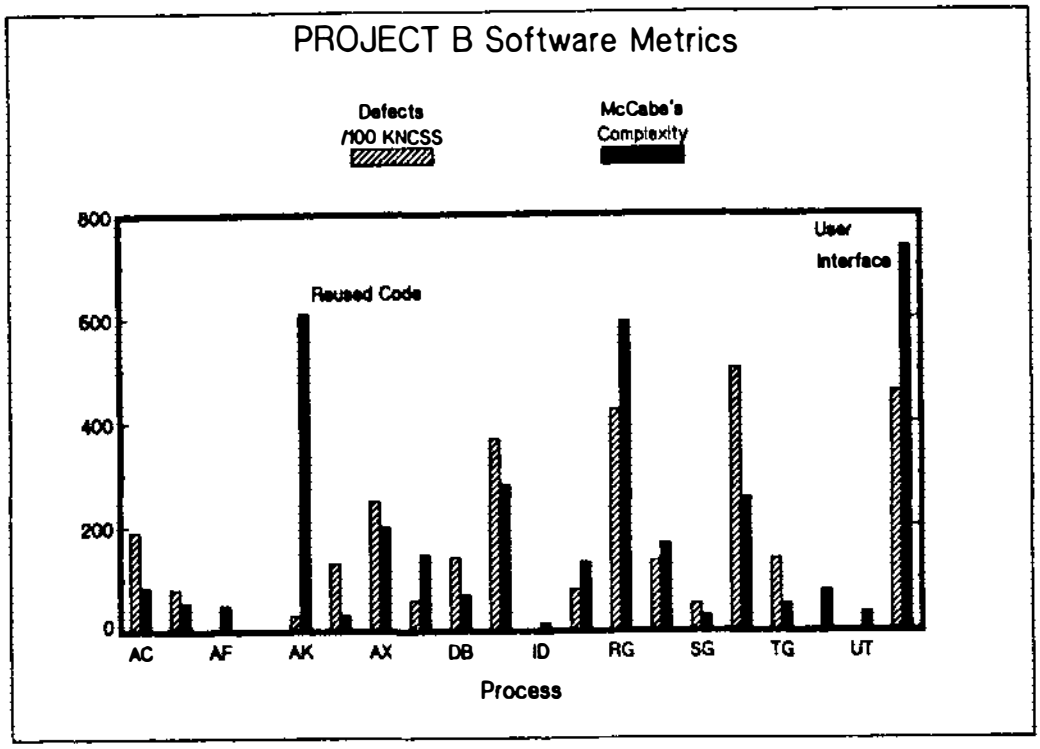


Figure 1. Complexity and defect rates for various modules of code.

2. A Toolkit

In order to address the issues of the development of reusable extensible user interface objects, the Digital Equipment Corporation and Hewlett-Packard Company have worked together to produce a toolkit. The core of this toolkit consists of the Xtk Intrinsics and a standard set of widgets.

The Xtk Intrinsics have been accepted as a standard approach to the definition and manipulation of user interface objects. These software objects are called **widgets.** The Intrinsics have been developed in a object oriented fashion using the C language. The Intrinsics define the root class of all software objects in the toolkit and a small set of standard methods for these software objects. The Intrinsics provide a consistent way to extend these methods to meet specific needs. Of great interest to the application programmer is the ability to manipulate these objects without regard to the complexities of window management under the X Window System™. The creation of windows and window hierarchies and window configuration are in large measure handled in a consistent manner by the Intrinsics.

An important goal of the Intrinsics is to leave user interface policy decisions to the widgets. In other words the Xtk Intrinsics attempt to provide a consistent framework in which to create a "look and feel." In terms of programmer productivity, this means that

once a programmer is familiar with the Xtk Intrinsics creating user interfaces of different looks and feels is merely a task of using different widgets. One need not learn yet another programming environment and library protocol.

The Xtk Intrinsics are by themselves insufficient for the application programmer. The framework for application interaction with widgets is defined, but the actual widgets are not. The Xtk Intrinsics have neither look nor feel. The creation of widgets which are visually consistent, bug free, and generally useful is a significant challenge.

Thus HP set out to create a set of widgets to address a broad range of applications. HP succeeded in creating two sets of widgets, of parallel functionality but different visual appearance. One set of these widgets and their source code has been contributed to the MIT X Consortium for nearly unlimited free use by the public at large. The other set is considered proprietary by HP. We will primarily concern ourselves with the first set.

The public widget effort at HP had a number of objectives. These objectives were of two types, quality objectives and feature objectives.

Key Quality Objectives:

- Product Quality - The widgets will be released as HP quality code. Another poor quality widget set would neither be a contribution to the UI field nor a good reflection on HP.
- A Test Suite - HP's toolkit effort is an open system effort. In order to the promote quality of these widgets as they are ported, we are providing a Test Suite. This test suite is a set of regression tests which will verify that a given port of the widgets is functional. As third-party ports of the widgets become more common, ISV's and their users can be assured of quality widgets.
- Documentation - The HP contributed widgets are well documented. This documentation consists of man pages for all widget classes, a tutorial, and a reference manual.
- A Broad Base Set of Widgets - Widgets which address the most common user interface paradigms. Users of these widgets will have to customize or produce a minimum of general purpose widgets.

Key Feature Objectives:

- Consistent Non-Proprietary Graphic Presentation - Due to the litigious nature of the UI business at present, it is imperative that the widgets bear little or no resemblance to user interface objects of certain companies. This was important for two reasons. Firstly, to protect HP from spurious lawsuits, and secondly, to protect users from spurious lawsuits.

- Widget Classing - Some user interface technologies provide the ability for the application end user to modify the application user interface. This facility may also be used to quickly modify the look or feel of an application without touching any source files.
- Keyboard Traversal - Touch typists strongly desire the ability to manipulate the user interface without reaching for the mouse. Forms style entry is desired, so that a user may move from input field to input field with a few keystrokes.

3. Widget Development

At this time (August 1988) the widget effort is generally considered to have been technically successful. Meeting the above objectives in a real world environment was challenging. The implementation of object orientation in the Xtk Intrinsics, and hence the widgets, is somewhat novel, and so a brief description of the implementation and HP's experience with it are the topics of this section. It is assumed that the reader has some familiarity with the basic concepts and terminology of object oriented programming. Readers who wish an in depth understanding of the intrinsic implementation are referred to Xtk Intrinsic documentation [3] and the intrinsic source code.

It is of primary importance to keep in mind that the Xtk Intrinsics were written in a standard dialect of C, and that the object orientation is a matter of style and organization. Widgets can be developed and implemented in a decidedly non-objective fashion, and although they might appear somewhat anomalous in either construction or interface, they can still compile and run rather nicely. However, the organization and the terminology of the intrinsics encourage one to think and design in object oriented terms.

An object within the Xtk Intrinsics is made up of two structures and a number of routines. The structures are the **Class** structure and the **Instance** structure. Both class structures and instance structures are made up of **parts.** Each part is the structure that is inherited from a superclass. Because C doesn't directly support inheritance, the inheritance chain must be explicitly given by the names and order of the parts in the class record. It is within the parts that the actual data is stored.

The class structure contains data common to all instances of this object this includes methods. Methods are kept within the class structure as pointers to functions. Generally the class structure contains things like instance defaults, class name, and methods. The instance structure contains data which is specific to each instance. Generally this is the data necessary to present the graphical image of the instance (e.g. the static text widget instance structure has a pointer to the string to be displayed for that instance).

The routines pointed to by these structures are standard C functions. For a given method the protocol of the function is fixed. For instance, all widgets have a resize method. All resize methods are functions taking a widget as a parameter and returning

void. An object may inherit methods from one of its superclasses by inheriting a function pointer or define its own function to handle the method. Generally an object must implement its own methods. An object may also create new methods.

The Xtk Intrinsics try to facilitate the object oriented model by completely hiding the class and instance data from the user (given proper style), and by facilitating such things as method inheritance. Unfortunately, there are some fundamental limitations which exist in the C language, and these tend to detract from the object orientation of the intrinsics style.

HP's experience with the Xtk Intrinsic implementation was mixed, but overall HP feels that the advantages far outweigh the disadvantages. The disadvantages stem primarily from unfamiliarity and the advantages come from being forced to use good software engineering practice.

Disadvantages:

- In the course of widget development, it became painfully obvious that there was significant cost associated with training programmers to follow the Xtk Intrinsic style. Otherwise proficient C programmers experienced several week learning periods before being productive widget implementors. Programmers' first widget took as much as twice the time as subsequent widgets of equivalent complexity.
- The Xtk Intrinsics have some non-obvious interactions with the widgets. These interactions have nothing to do with object orientation, they are simply the nature of the method semantics. In short, the intrinsics are not perfect yet.
- Because of implementer familiarity with C, strict adherence to the object oriented style was hard to achieve. Somethings just seemed easier to do and understand in a procedural fashion. A specific example is structure access. All widgets implement a method for exporting public data, however because of the availability of the structure definitions it was easy to just read the data directly from the structure. Complicating the matter is the fact that direct structure reads can be well defended from the perspective of application performance.

Advantages:

- The Xtk Intrinsics went through one major revision during the HP Widget development. This revision was accommodated mid-stride by the development team at the cost of less than one week of calendar time. The program structure encouraged by the widgets localized all changes, and allowed any developer familiar with the intrinsics to work with any widget.
- New widgets were easily created by using old ones as templates. The structure of all widgets is much the same. Certain design decisions are already made by the intrinsics,

so the developers did not have to design from scratch each time.

- During the course of development several different visual styles were experimented with. The object structure made modification easy by localizing changes and facilitating code reuse (by method inheritance).

4. Widget Use

In this section we discuss the widgets which have been made available for use and the experience of our users. A complete explanation of the widget set is not within the scope of this paper. What follows is a brief description of the widgets so that the reader may gain an appreciation of the scope of the widget set and the interactions supported. For detailed information the reader is encouraged to read the HP X Widget documentation itself.

The widgets can be classified into two major groups. The two major groups being primitive widgets which perform some sort of interactive task, and manager widgets which control the layout of other widgets.

The primitive widgets are:

- StaticRaster - Displays an uneditable image. May be selected.
- ImageEdit - Displays an image in a magnified fashion. Allows the user to manipulate a raster.
- Toggle - Supports a button type interaction with the selection indication separate from the button label. A button type interaction is a select/release action where select and release is usually done with the mouse.
- PushButton - Supports button type interaction with the selection indication including the button label.
- StaticText - Displays an uneditable string. Understands string wrapping and may be selected.
- TextEdit - Displays an editable string.
- Valuator - Associates some amount of screen space with a quantity. By moving the "slider" the user can indicate an amount.
- ScrollBar - A valuator with "arrow" buttons to facilitate the movement of the slider.
- WorkSpace - A blank-canvas-widget so to speak, which allows the application developer to easily and quickly provide specialized graphical functionality within the paradigm defined by the Xtk Intrinsics.

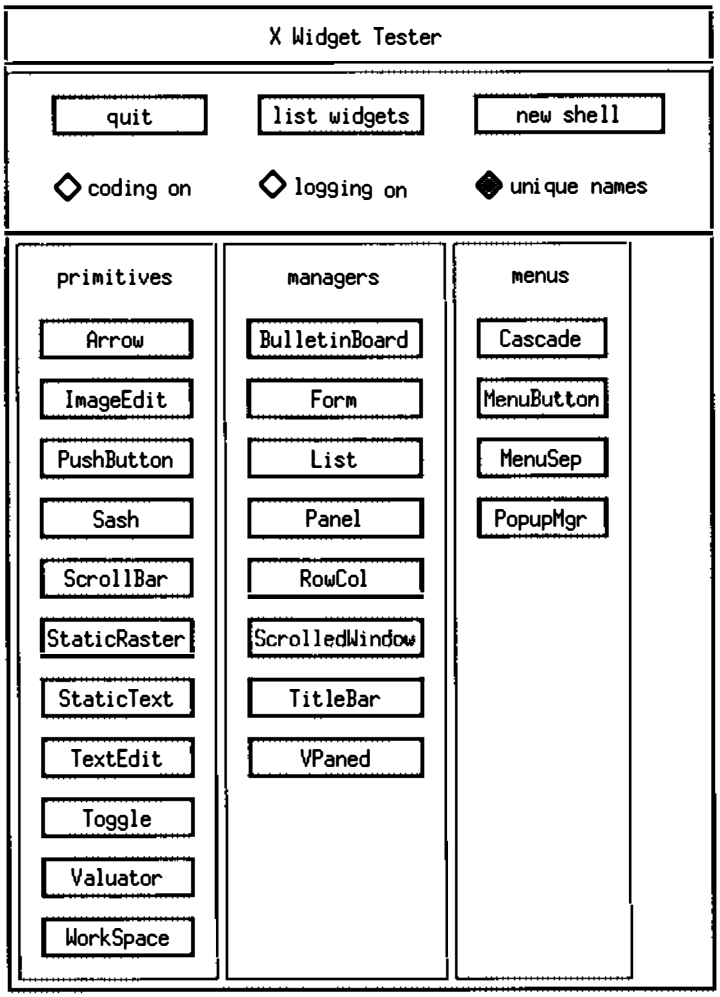


Figure 2. A display illustrating static text, toggle and pushbutton widgets.

The manager widgets are:

- Bulletin Board - Allows arbitrary placement of widgets.
- Form - Supports relative layout of widgets; widgets are placed relative to other widgets.
- List - Provides the ability to select arbitrary groups of arbitrary widgets.
- Panel - Provides a default layout for application level windows.
- Row Column - Neatly orders groups of widgets into rows and columns.
- VerticalPanedWindow - Allows the management of arbitrary widgets in a vertical fashion.

- ScrolledWindow - Allows arbitrary widgets to be scrolled within a window.
- TitleBar - Manages horizontal layout of arbitrary widgets.
- Menus - Both Pop-Up and Pulldown type menus are supported in seamlessly interchangeable manner. Interchangeability is achieved through the use of widget classing.

A sample of display using these widgets is found in figure two.

These widgets group support all of the common user interactions of general purpose software. The primitive widgets support the gathering and communication of user actions to the application, and the manager widgets provide consistent reliable behavior during arbitrary resizing operations.

Widgets and the Xtk Intrinsics have not been available to programmers for very long, so their track record is not established yet. Despite this recent availability, there have been a number of uses of the widgets by applications programmers especially within the Hewlett-Packard Company. Unfortunately, these applications are (or will be) new or internal products, and since it is not HP policy to publicize such products in the forum of technical papers, these applications can only be referred to in a general way.

The widgets have been applied to existing programs. A number of keyboard/terminal based applications have been move to the X Window System using the toolkit. These applications are predominantly file manipulation programs. Our users report that creating the user interface was quick and straight forward.

The widgets have also been used for rapid prototyping. As part of the feasability study for a new product, a user interface was built from the widgets. The engineer of this project reported that his investment in this prototype was approximately twenty hours. He further stated that without the widgets there would have been no way for him to have provided a sample interface for evaluation. The design effort at the Xlib level would have been too costly for a prototype.

Several new products are currently being developed using the toolkit. In these cases no rapid prototyping has been done. These products are, however, follow-on products to some internal tools previously built using Xlib (protocol version 10). The development team reports significant productivity increases due to the application of the toolkit to their user interface needs.

All of our users have encountered some shortcomings with the current toolkit. Widget shortcomings are in the area of generality. The widgets were developed with the widest possible audience in mind. Despite this, some significant user needs were overlooked particularly in terms of inter-widget interaction. Xtk Intrinsic shortcomings have been in the area of performance (nothing ever goes fast enough), and completeness. There are some apparently obvious modes widget control which are not well supported.

5. Conclusions

As widget developers, we learned a great deal about the relative value of the object oriented environment in which we worked. There were two drawbacks to the object oriented environment. First, that it was implemented in straight C which does not support the complete object oriented paradigm very well; breaking the paradigm was very easy, and in some cases encouraged, and some needs, such as object initialization, are simply lacking.

Secondly, the newness of the approach was costly for widget developers. In order to begin to program, a tremendous number of details in style and interaction had to be understood. Step-wise widget development was not feasable. The cost of developing proficiency at widget writing was much larger than originally anticipated.

Despite these drawbacks, we found the object orientation to be a valuable approach overall. It reduced the impact of the tight coupling between the widget implementation and the intrinsic implementation. This factor alone balanced the schedule impact of learning the environment. The object orientation allowed rapid enhancements as we refined our design decisions, and it encouraged code reuse between developers.

Feedback from the use of the widgets seems to follow a few general trends. The toolkit to date has shown itself to be very effective in the area of rapid prototyping. The nature of the communication between the application code and the widget code makes prototype development straightforward and fast. Widgets show great promise for developing applications both for their prototyping ease and for the ease with which visual styles can be customized at runtime.

6. Implications

A clear implication to programmers who are developing interactive applications under the X Window System™ is that there is a useful standard which is freely available. Interaction prototypes can be quickly developed and shown to prospective users. As work develops programmers can count on the interface behaving in a reliable and consistent manner.

More generally, we were able to make a couple of observations regarding software development in general. Object orientation encourages a looser coupling between development teams than traditional development methodologies. A development effort with two different companies in two different states, not communicating for months at a time was successfully completed. Object orientation also encourages code reuse, both at the compiled code level through superclasses, and at the source code level by modifying sibling subclasses.

REFERENCES

[1] **Xlib - C Language X Interface, X Window System, X Version 11, Release2** by Jim Gettys, Ron Newman and Robert W. Scheifler. Copyright 1988 Massachusetts Institute of Technology and Digital Equipment Corporation. No publisher.

[2] *Timely Software Defect Removal Using McCabe's Automated Code Analysis Tool* by Jack Ward, published in **Software Engineering Productivity Conference 1988 Proceedings** an HP internal publication, copyright 1988 Hewlett-Packard Company.

[3] **X Toolkit Intrinsics - C Language X Interface, X Window System, X Version 11, Release 2** by Joel McCormack, Paul Asente, and Ralph R. Swick. Copyright 1988 Massachusetts Institute of Technology and Digital Equipment Corporation. No publisher.

Tina: A Facility for Assisting With Unit Testing

Richard D. Sidwell
Gordon W. Blair
Rockwell International Corporation
3370 Miraloma Ave.
P.O. Box 4192, Mail Code GA-21
Anaheim, California 92803-4192

Abstract

Tina is a system to assist software developers with unit testing. Rather than being based on a particular language, or being independent of any programming language, Tina is a set of extensions which can be applied to most programming languages that make writing tests easier. The main construct is the check-using statement, which allows easy specification of the most common software test cases. Additional constructs allow including other tests in the test report and provide control over the testing process.

Biographical Sketches

Richard D. Sidwell leads an Independent Research and Development group in advanced software development at Rockwell International Corporation. His current research interests are software testing, software environments, and concurrent systems. He received his B.S. in computer science from Utah State University and his Ph.D. in computer science from the University of California at Irvine in 1987. Rick is a member of the Association for Computing Machinery.

Gordon W. Blair works for Rockwell International Corporation where he has been involved with several projects pertaining to software reuse. His current interests are software testing, software reuse, and software environments. He received his B.A. in mathematics and computer science from Point Loma College in 1984.

Tina: A Facility for Assisting With Unit Testing

Richard D. Sidwell
Gordon W. Blair

Introduction

One of the most tedious and time consuming parts of software development is that of software testing. Many people have attacked this problem by producing software testing tools. Previous methods can be divided into three classes: utilities designed for system testing, systems for testing program units written in specific languages, and language independent systems. Utilities designed to test entire systems use operating system features to intercept outputs intended for external devices and to simulate inputs from such devices. Some examples of system testing utilities are the VAX Test Manager by Digital Equipment Corporation [1] and the Automated Software Test Facility by IBM [2].

Several testing systems have been developed for unit testing of programs which take the language in which the programs are written into consideration, such as TPL/F (Fortran Test Procedure Language) by General Electric [3] and UDT (Unit Debugging and Testing) by Intel [4]. These systems allow test engineers to take advantage of programming language constructs as they write their test cases, but such systems are only applicable to one specific language—the constructs are not always easily applied to other languages.

Other systems are language independent, such as AUT (Automated Unit Test) by IBM [5] and TCL (Test Case Language) by Intel [6]. These systems work with any source language, but do not let testers take advantage of the advanced features of modern programming languages. These systems also require programmers who test their own units to use two different languages: one for writing programs and one for writing test cases.

Tina is a new system that combines the power and ease of use of language dependent systems with the portability of language independent systems. It consists of a set of extensions which can be applied to most programming languages that facilitate writing software tests. Tina is language dependent in the sense that full use of all programming language constructs is allowed, but the extensions that comprise Tina are language independent, and can be applied across programming languages. The Tina processor translates these extensions into valid programming language statements, which can be compiled with the normal compiler.

Tina was designed to test software units—program parts which are used by other program parts to perform specific functions. Tina does not facilitate the testing of entire systems, which communicate with external devices and which generally require different testing techniques. Thus Tina complements rather than competes with system level testing utilities.

The Tina Language

The Tina language consists of three statements that are allowed in addition to the normal statements of the programming language being used: check-using statements, report statements, and Tina directives. The check-using statement provides a convenient way to specify a series of tests which Tina is to perform. For applications where this statement is too constraining for needed tests, arbitrary code in the target programming language can be

used in conjunction with the report statement, which is used to communicate the results of such tests so that they can be included in the test report. Finally, directive statements specify some parameters for testing, such as the precision of floating point number comparison and the amount of time that Tina should wait before assuming that some procedure is in an infinite loop and should be aborted.

The Check-Using Statement

A study of common software tests showed that most tests fit the following criteria:

- The function being tested is called multiple times,
- The only changes between calls are the inputs and outputs, and
- Success of each test case is determined by comparing the expected and actual results for equality.

This observation has led to the development of Tina's check-using statement. This statement allows programmers to provide a test specification with inputs and expected outputs for a number of test cases in a compact manner. The expected outputs are automatically compared with the actual outputs.

To better understand the check-using statement, consider the following example. A procedure called `AddOne` is to be tested whose only effect is to add one to the global integer variable `Counter`. A simple check-using statement to test this procedure is:

```
check {AddOne();} using
  [Counter => Counter],
    [4=>5], [0=>1], [-7=>-6];
```

Between the check and the using is a statement (or statement list) which calls the unit being tested—in this case, a simple call to `AddOne`. Following this is a list of items in brackets. The first is called the *template*, which identifies the variables involved in the test. The input variables are separated from the output variables by an arrow, =>. In this case, there is one input variable, `Counter`, and one output variable, also `Counter`.

The items after the template are test cases. For each test case, the input variables specified in the template are given the input values specified in the test case (to the left of the =>), and the statement is executed. The output variables are then compared with the expected output values from the test case (to the right of the =>) and the result reported. The next test case is then executed in the same way.

Since most functions which are tested act on parameters rather than global variables, it is useful to allow new variables to be declared inside check-using statements. Since a goal of Tina is to make it as easy as possible for a programmer, fluent in some programming language, to use Tina, these local declarations have been structured to be similar to other variable declarations in the language being used. The declarations are given in the template. For example, a check-using statement to test a function `PlusOne` written in Ada, which returns the result of adding one to its parameter is:

```
check {B := PlusOne(A);} using
 [A:Integer => B:Integer],
   [4=>5],[0=>1],[-7=>-6];
```

It is also possible to mix local and global variables in the same check-using statement. Consider, for example, a check-using statement to test a C function AddUp which adds its integer argument to the global variable Sum:

```
Sum = 0;
check {AddUp(Num);} using
  [int Num => Sum],
    [3=>3],[7=>10],[-1=>9],[0=>9];
```

The global Sum is initialized to zero before the check-using statement, which only modifies Sum using AddUp. Using global variables in this manner is possible since the tests are executed in the order specified in the check-using statement. Note that since C is used, the declaration for the local variable Num used the C variable declaration syntax, with the type before the variable name. Another way to test AddUp which is not quite as elegant, but illustrates some check-using statement features is:

```
check {AddUp(Num);} using
  [int Num; Sum => Sum],
    [3;0=>3],[7;3=>10],[-1;10=>9],[0;9=>9];
```

Rather than letting the global Sum keep its value from call to call, this statement explicitly initializes Sum each time. This example also illustrates how several inputs are specified in a check-using statement.

Many languages, such as Ada, have the ability to handle exceptional conditions with special language features. The following example, a test for a square root function written in Ada, illustrates two features of check-using for testing exceptions:

```
check {B := Sqrt(A);} using
  [A:float => B:float],
    [4.0=>2.0], [1.0=>1.0], [2.0=>1.4142136],
    [-3.0=>?=>numeric_error];
```

The question mark in the output field of the last test case indicates that output is not known—when an Ada function raises an exception, it does not return a value. Such "don't care" values may be specified in place of any input or output. A don't care input will not change the corresponding variable; it will keep the value last assigned to it. A don't care expected output means that the value assigned to that variable should not affect the success or failure of the test case.

An expected exception is specified by placing its name after a second arrow, after the output values. Thus in the square root example, attempting to take the square root of -3 will raise the numeric_error exception.

The last feature of check-using statements is the ability to embed host language code between the test cases. This is useful for performing special functions between some of the tests. Consider, for example, the following statement for testing a Cartesian coordinate scaling procedure Scale, written in Ada, that converts points from one coordinate system to another. The scaling factor is specified by calling the procedure Set_Scale, which sets the source coordinate system point passed to it to correspond to (1.0,1.0) in the target coordi-

nate system. The type `Point` is a record containing two floating point numbers, the X and Y coordinates of the point. The initial scaling factor is (1.0,1.0), no scaling. Calls to `Set_Scale` are embedded in the following check-using statement to provide a more complete test of `Scale`.

```
check {New := Scale(Old);} using
  [Old:Point => New:Point],
    [(0.0,0.0) => (0.0,0.0)],        -- Three tests using
    [(5.6,9.3) => (5.6,9.3)],        -- default scale factor
    [(-3.2,4.1) => (-3.2,4.1)],
   {Set_Scale((-2.0,4.0));}          -- Set a new scale factor
    [(1.0,4.0) => (-0.5,1.0)],       -- Three tests using the
    [(-5.2,-8.4) => (2.6,-2.1)],     -- new scale factor
    [(7.6,1.0) => (-3.8,0.25)],
   {Set_Scale((0.5,1.0));}           -- Try one more
    [(0.0,0.0) => (0.0,0.0)],
    [(1.0,1.0) => (2.0,1.0)],
    [(-4.3,-9.2) => (-8.6,-9.2)];
```

Note that the points are specified using Ada aggregates (the values of the record are listed in order between parentheses). Since they are so useful for testing, implementations of Tina in languages which do not normally support aggregates can have them built into check-using statements. For example, the C language provides only limited support for aggregates—as initial values for static arrays and structures. The C version of Tina allows them to be placed in check-using statements as well.

Actual use of check-using statements to date has shown that they are very useful for specifying most test cases that arise in practice. They provide a concise representation of the information needed to specify test inputs and expected results.

The Report Statement

The report statement is a Tina construct that is available when the check-using statement is not appropriate for a particular test. While check-using is quite flexible, there are many cases where it is not appropriate. In such cases, arbitrary code in the target language may be used to write the test; the report statement then provides the ability to have the input and output data for such tests formatted and included in the final test report.

The report statement is geared toward an individual test case. Typically, the test would be performed by application language statements, followed by report statements to denote the input, expected output, actual output, and the result of the test case. Some examples of when the report statement would be useful (i.e., that can not easily be specified with check-using) are:

- Tests which are not for exact equality. The exact value may not be known. Or the result is a record and only some of the fields are significant. Or statistical analysis over many unit calls is required.
- Tests for which the inputs are difficult to specify. Perhaps complex recursive data structures are required. Or a large array whose values generally remain the same, but with irregular changes between test cases.

- A repetitive test is needed, such as for stress and capacity testing. For example, to test that a stack which should hold 1000 elements actually does, 1000 elements must be pushed onto it and popped back off. Doing this with a check-using statement would require these elements to be listed individually—a for-loop would be more compact and readable, as well as easier to write.
- Interleaved calls to different functions are needed. For example, to test a stack, some elements are pushed onto it, and later popped off of it. The calls to push and pop are interleaved in an arbitrary order. The check-using statement is designed to test only a single function at a time.
- "Sanity checking" of global variables between other tests is needed. This is often done to ensure that they remain in a consistent state.

The report statement comes in four flavors: one to report inputs (or more generally, the parameters which make this test case unique), one to report outputs (both expected and actual), one to report whether the test was successful or not, and one to report that the test was not valid for some reason (such as taking too long, requiring the test case to be aborted). Here are some example report statements:

```
report input A;
report inputs "push",X;
report output Expected:Actual;
report (Expected = Actual);
report invalid;
```

The keyword after the word report specifies the meaning of the rest of the statement. The meanings of most of these should be obvious; the only one that does not explicitly state its purpose is reporting the success or failure of a test—this is done by reporting a boolean expression; the test is successful if the expression is true, and unsuccessful if false.

As an example of the use of the report statement, consider the following program fragment for testing an Ada package which implements a double ended queue (a combination of a stack and a queue, in which elements can be pushed on or popped off either the front or the rear):

```
Push_Front(4);                      -- push 4 on the dequeue
report input "Push_Front",4;        -- report what we did
report True;                        -- pushing is successful

Push_Front(10);
report input "Push_Front",10;
report True;

Push_Rear(-7);
report input "Push_Rear",-7;
report True;

X := Pop_Front;                     -- pop from the front
report input "Pop_Front";           -- report what we did
report output 10:X;                 -- report the outputs
report (X=10);                      -- successful if 10 popped
```

```
X := Pop_Front;
report input "Pop_Front";
report output 4:X;
report (X=4);

X := Pop_Front;
report input "Pop_Front";
report output -7:X;
report (X = -7);
```

Note that the first input value is not really an input, but the name of the procedure being called for the particular test case—the purpose of reporting inputs is to identify the particular test being performed for the test report. Also note that `Pushes` are always successful, and that this fact needs to be reported using the statement "report True," which indicates the (successful) end of the test case.

The Tina Directive

The third and final kind of statement in Tina is used to specify options for different tests: the Tina directive. They can appear anywhere another statement can appear. The current version of Tina includes four directives:

```
tina limit(x);

tina precision(d);

tina title("Test title");

tina labels("Input 1","Input 2" => "Output 1", "Output 2");
```

The limit directive is used in conjunction with the check-using statement. It specifies a time limit (in seconds) for the execution of each test case of the next check-using statement. If this time limit is exceeded, Tina assumes that the procedure being tested is in an infinite loop and aborts it. The test is reported as "invalid."

The precision directive is also used with the check-using statement and specifies the precision (as a number of decimal digits) with which floating point comparison should be done. Tina does not do exact equality testing with real numbers since no computer can represent all real numbers exactly. The specified precision stays in effect until changed with another Tina precision directive.

The title directive is used to specify a title for a set of test cases. It can be as simple as "Test 1" or more complex, such as "Test of the square root routine." It is especially useful when project requirements require a special identification convention for the tests (e.g., "Test DTS032-TXO740367").

The labels directive is used to label the input and output values of the tests. If no labels directive is given for a check-using statement, the variable names associated with each input and output will be used. When using report statements, a labels directive should always be given to specify the meanings of the input and output values which are reported. The order of the labels corresponds to the order in which the values are reported.

A Tina Example

To see how these Tina constructs are embedded in a standard programming language, consider the following example of a complete Tina program to perform a very rudimentary test of an Ada square root routine:

```
with Sqrt;
procedure Test_Sqrt is
-- Tina program to test Ada square root routine

begin

tina precision(5);
tina title("Test of Sqrt");
tina labels("X" => "Sqrt(X)");
check {B := Sqrt(A)} using
  [A:float => B:float],
    [4.0=>2.0], [1.0=>1.0], [2.0=>1.4142136],
    [-3.0=>?=>numeric_error];


end Test_Sqrt;
```

The first few lines are Ada statements which get the function under test, `Sqrt`, from the library and declare the name of the test procedure, `Test_Sqrt`. The first three Tina statements are directives which define the precision, title, and labels for the test. Next comes the check-using statement that was described earlier to test the square root function. Finally, an Ada end appears to match the earlier begin.

A similar procedure to test a square root program written in C would be:

```
main()
/* Tina program to test C square root routine */

{
tina precision(5);
tina title("Test of sqrt");
tina labels("x" => "sqrt(x)");
check {b = sqrt(a);} using
  [float a => float b],
    [4.0=>2.0], [1.0=>1.0], [2.0=>1.4142136];
}
```

The differences in the check-using statement are to make it conform to the syntax of the C language rather than Ada: the declarations are different, and the statement to be tested (`b = sqrt(a);`) uses the C assignment operator "=" rather than the Ada ":=" operator. Of course, the structure of the program itself is entirely different in C—braces are used rather than begin-end, the procedure heading is different, no "with" is required in C, and the comment convention is different.

Tina Implementation

In breaking down the design of Tina, the focus has centered around three areas: the infusion of the Tina extensions into the programming language, the execution of Tina programs, and the report generation of the test results. This structure is shown in Figure 1. A test program is written in the Tina language, which is a general purpose programming language with the Tina extensions described previously. The Tina processor transforms the Tina constructs into valid programming language statements, compiles the resulting program, and links it with the unit to be tested. The resulting program is then executed and the test results formatted with the Tina postprocessor. Optionally, an execution profile of the unit under test can also be generated in order to provide coverage data in the test report.

Tina Language Considerations

The main concern for adding the Tina extensions is making them blend in with the rest of the language. We have already seen how, for example, the declarations of local variables in the templates of check-using statements in C and Ada change to match the language being used. But Ada and C statements have basically the same structure: freely formatted statements terminated with a semicolon. Tina statements added to a language with a differ-

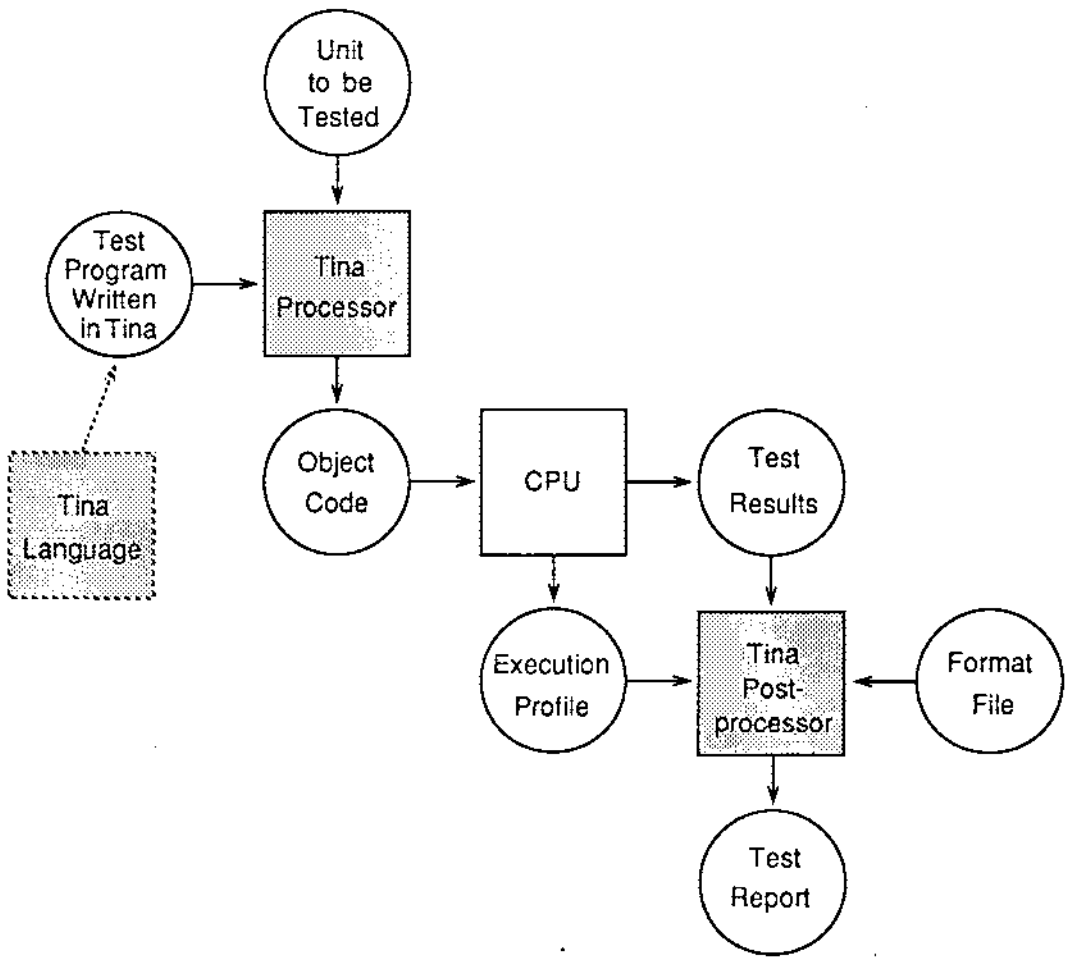


Figure 1. Components of the Tina System.

ent statement structure would need to conform to the underlying language. Consider, for example, the FORTRAN language, where statements are line oriented. A check-using statement might have the following appearance:

```
CHECK
   B = SQRT(A)
USING
   [REAL A => REAL B]
     [4.0 => 2.0]
     [1.0 => 1.0]
     [2.0 => 1.4142136]
```

Since each test case appears on a separate line, there is no need of a comma to separate them. There is also no need for a final semi-colon; the first line that does not begin with a left bracket implicitly ends the check-using statement.

A final consideration for Tina language features is the ability to specify some structured data types, such as arrays and records, in the check-using statement. Some languages, such as Ada, feature aggregates as part of the language, thereby easily allowing these to appear in check-using statements. Other languages do not normally allow this, but since they are very useful for writing tests, they should be considered for inclusion. This adds complexity to the Tina processor, but it is probably worth the effort.

Tina Processor

After a test program has been written, it must be compiled and linked to the module being tested. Since the test program contains Tina extensions, the standard compiler will reject it. The purpose of the Tina processor is to convert this test program with special Tina statements into a legal program in the application language, which can then be compiled with the standard compiler, linked with the module being tested like any other program which uses that module, and run like any other finished program on the system.

When the test program is thus run, it places the results of the test in a place where the postprocessor can access them. In a system with output redirection and piping, like UNIX, the results can be written to the standard output, which can be piped into the Tina postprocessor directly, or redirected to a file for later formatting. In other systems, the results can be placed in a file that the postprocessor can then read to generate a test report.

Just as the Tina language extensions should blend in with the underlying language, the Tina processor should be integrated into the development environment. The question of where to direct the test result is not the only issue which must be addressed. The command used to invoke the Tina processor should be similar to the command used to compile an ordinary program. Similar options should exist, for example, to enable debuggers to be used and keep intermediate files as well as to link in library units. Such options can be implemented by simply passing them to the real compiler after the processor is finished translating the Tina constructs in the test program.

A problem arises with languages such as Ada which use libraries to store information about separately compiled units. The Tina processor needs information about the types and variables used in check-using and report statements. If the library is stored in a proprietary format, this information will be difficult to obtain. There are several ways to solve this

problem The user could be required to specify where the original source code for the type definitions is, or to run units through Tina in the same order they would normally be run through a compiler. These methods are general enough to work for any language, but preclude using precompiled units for which the source is not available.

An alternative is to have the Tina programmer write additional code that would otherwise be generated by the Tina processor—code to output and compare custom data types. This is an acceptable alternative for Ada, since the ability to overload functions allows the programmer to simply write a number of procedures with the name "put" that output each structured data type used in a Tina statement; comparison in Ada is predefined for most data types. Libraries are not a problem in languages such as C, which simply include "header files" containing the type and global variable definitions.

Another subject which must be addressed with regard to the Tina processor is the kind of potential restrictions imposed on the underlying language when writing Tina programs. Ideally, there would be no restrictions, but such a processor would probably have to parse the entire program, instead of simply looking for type definitions and Tina statements. The only restriction required by current Tina implementations is that variables used in Tina statements have only one type declaration—there should not be several variables of that name in different scopes of the program, which would require Tina to keep track of the block structure. This restriction is not too serious, since it is poor programming practice to hide the definitions of important global variables with local declarations.

Tina Postprocessor

The Tina postprocessor performs the function of generating a report for the user displaying the testing information associated with the execution of the test. The Tina postprocessor displays the testing information in two modes, the single test case mode which focuses in on all elements associated with a particular test case, and the table mode which emphasizes a particular element tracked in each individual test case of a test.

The single test case mode groups all the information related to a single test case and displays it so the user can quickly analyze the outcome of the particular test case (see Figure 2). Indentation is used to help distinguish groups of data (e.g., structures in "C" and records in Ada). The expected and actual outputs are separated so the user can focus on the type of outputs as a whole and not just on individual values. This mode of reporting provides the user the ability to easily see what went wrong when a test case failed.

The table mode gives the user a test perspective of how each individual element changed according to the boundary value specified for each test case (see Figure 3). The individual test case value is stressed more in this mode than how it is grouped according to the particular data structure (e.g., array). The multiple tables within the input and output categories are arranged in sequential order according to how they were specified in the Tina program. This mode of reporting is geared toward giving the user the ability to track individual changes in order to check for testing completeness and specific problems.

In actual use, the Tina postprocessor is typically used to first print out the elements in the single test case mode and follow that with the same elements in the table mode. To allow the user to suppress either the single test case or table mode, the postprocessor first parses a predefined format file to identify options specified by the Tina user. The options per-

mit suppressing the above mentioned modes as well as setting up certain limits in terms of how many entries are desired in a table, if truncation of the labels or values can take place. Finally other display format options relating to the displayed values are specified.

Another display option of the postprocessor permits consistency checking. Through incorrect use of the report statement, Tina users can be inconsistent in their specification of test case information which should be constant for a particular test. The postprocessor by default warns of such inconsistencies. However, this consistency checking option can be suppressed in the postprocessor format file if for some reason (e.g., Tina is being used for debugging rather than testing) the warnings only clutter up the test data.

After reading the format file, the Tina postprocessor accesses report data from the program output generated by the Tina processor. There are predefined formats established between the Tina processor and the postprocessor reflecting the kind of information Tina processes (i.e., input and output values, input and output labels, test titles, and test case results). These formats also identify the application language's types (e.g., numeric, string, array, or record), so a particular report formatting pattern can be followed.

```
     Test Title:  Outline for test case results with Tina "C"

TEST CASE #1
  INPUT:
    scalar_value       = 16
    array_value(0,0)   = 0
    array_value(0,1)   = 1
    array_value(1,0)   = 2
    array_value(1,1)   = 3

  OUTPUT:
Expected Output:
  structure_variable
    first_field    :
      array(0)     = 10
      array(1)     = 20
    second_field   :
      scalar       = "this is a string"
      length       = 16

Actual Output:
  structure_variable
    first_field    :
      array(0)     = 10
      array(1)     = 20
    second_field   :
      scalar       = "this is a string"
      length       = 16

                              TEST CASE #1  result => pass
```

Figure 2. The Single Test Case Mode Format

The postprocessor has been separated from the processor for two reasons. The first is to help improve Tina's portability to other software programming languages. The second reason is to make the report generation facilities more adaptable to change. The Tina processor must be rewritten for every new language implementation, but if the interface to the postprocessor remains the same, the postprocessor only has to be rewritten if placed on a machine that does not support the language in which the postprocessor is implemented. However, the postprocessor will over time be enhanced and modified as well as ported to languages where the displayed format of current data type groups (e.g., the C language type groupings of numeric, string, array, or structure) does not adequately display results to the Tina user.

If the postprocessor cannot adapt to the needs of the user, or a forms generating software package becomes available that better fits the requirements of the user, the file

```
     Test Title:  A table of test results

                     INPUTS

test case     struct.first_field   struct.second_field
   #1               40                    41
   #2               30                    31
   #3               -1                    0


test case     struct.third_field
   #1            "string field"
   #2          "it could contain"
   #3                "names"

                    OUTPUTS

test case                      array(1)

                   (actual)             (expected)

   #1               350                   350
   #2               28                    28
   #3               -57                   0

test case                      array(2)

                   (actual)             (expected)

   #1               222                   222
   #2               34                    34
   #3               -10                   0

          Results

test case       result
   #1            pass
   #2            pass
   #3            FAIL
```

Figure 3. The Table Mode Format

interface between the processor and the postprocessor makes using a different postprocessor for Tina easier.

Future Work

Although Tina as currently implemented is a useful tool for software testing, there are several issues which should be addressed in future versions. One is the use of pointers, the only common programming language feature not currently supported by Tina. Pointers are usually combined with record types to form potentially complex, dynamic structures, which are difficult to deal with in a general way. There are two main problems. The first is representing complex recursive structures in a textual form. A LISP-like format can be used for trees, but there is no concise, readable format for structures with shared substructures or cycles. The second problem is comparison of two structures; the definition of equivalence for dynamic structures varies from application to application. The problems of shared substructures aside, are the following two trees equivalent? Only if order is not important.

Another consideration for future enhancement of Tina is the addition of new statements. The check-using statement is applicable to the majority of software tests which are needed, but there are other common test situations for which it is difficult to use. One of these is when an abstract data type is to be tested, for example an Ada complex number package or a set of C routines to implement a stack. The check-using statement is not appropriate because it calls the same routine for each test case; tests of abstract data types need to call different procedures to make sure that they interact properly together. A statement to help test units like this would make a useful addition to Tina.

The current implementation of Tina performs only black-box testing—the unit under test is not modified for the test. Extensions to support white-box testing is another feature to consider in future versions of Tina. This would allow access to variables which are local to the unit, and permit forcing certain paths to be taken, permitting better test coverage without having to calculate the input values needed to execute little-used paths.

Conclusions

Four underlying goals for Tina were to first evolve a general unit testing paradigm; second, to implement easy to use extensions for this paradigm within the targeted programming languages; third, to provide insights for the user to permit the incorporation of unit testing as an integral part of the development process; and fourth, the integration of Tina into the software development environment. The attributes that characterize Tina are simplicity and flexibility in both its implementation and use. Tina finds its niche in software testing by providing a fast and easy way to perform software unit testing without significant time investment in learning the notation. The belief is Tina provides a model that facilitates software testing across applications and software languages which has significant utility as presently implemented and significant potential in future applications.

References

[1] Digital Equipment Corporation, *VAX DEC/Test Manager Reference Manual*, 1985.

[2] International Business Machines Corp., *Automated Software Test Facility User's Guide*, Order number SH21-0001, 1986.

[3] D.J. Panzl, "Automatic Software Test Drivers," *Computer*, Vol. 11, no. 4, April 1978, pp. 44–50.

[4] B. Albiso, M. Motasim, and C. Thomas, "UDT: a Tool for Debugging and Testing Software Units," *Proceedings of the Fourth Annual Pacific Northwest Software Quality Conference*, 1986, pp. 118–135.

[5] International Business Machines Corp., *Automated Unit Test (AUT) Program Description/Operation Manual*, IBM Installed User Program Number 5796-PEC, August 1975.

[6] A. Jagota and V. Rao, "TCL and TCI, A Powerful Language and an Interpreter for Writing and Executing Black Box Tests," *Proceedings of the Fourth Annual Pacific Northwest Software Quality Conference*, 1986, pp. 147-155.

Testing

"Experimental Results of Automatically Generated Adequate Test Sets"
Richard A. DeMillo, Purdue University; A. Jefferson Offutt VI, Clemson University

"Using Data Flow Analysis for Regression Testing"
Thomas J. Ostrand, Siemens Research and Technology Laboratories;
Elaine J. Weyuker, Courant Institute of Mathematical Sciences, New York University

"Testing Shared-Memory Parallel Programs"
Andrew H. Sung, New Mexico Tech

Experimental Results of Automatically Generated Adequate Test Sets

Richard DeMillo
Professor and Director
Software Engineering Research Center
Department of Computer Sciences
Computer Science Building
West Lafayette, IN 47907
rad@Purdue.edu
(317) 494-7823

Jeff Offutt
Assistant Professor
Department of Computer Science
Clemson University
Clemson, SC 29634
ofut@Clemson.edu
(803) 656-3444

KEY WORDS

software testing, test data generation, mutation analysis, Mothra, constraints

ABSTRACT

A test case generator that is part of the Mothra project creates mutation-adequate test data. The test cases are generated from mathematical constraints that describe test cases. The experiments validate the technique and implementation.

BIOGRAPHICAL SKETCHES

Richard A. DeMillo is Professor of Computer Science and Director of the Software Engineering Research Center at Purdue University. In his current position, he teaches and conducts research in computer science and software engineering. He also directs the operation of The Software Engineering Research Center, a National Science Foundation Industry-University Cooperative Research Center. This interdisciplinary research center consists of faculty, professional staff and graduate research assistants who conduct basic and applied research and carry out technology transfer projects on behalf of its Government and Industry sponsors and affiliates.

A. Jefferson Offutt is Assistant Professor of Computer Science at Clemson University. He graduated with a BS in mathematics and data processing in 1982 from Morehead State University. He earned the MS degree from Georgia Institute of Technology in 1985 while teaching courses and working on the Software Test and Evaluation Project. Eventually, he joined the Mothra project in Georgia Tech's Software Engineering Research Center, wrote a Fortran interpreter, and finally began work on his doctoral research. His doctoral dissertation is titled "Automatic Test Data Generation" and was completed in August of 1988.

Experimental Results of Automatically Generated Adequate Test Sets*

Richard A. DeMillo
Purdue University

A. Jefferson Offutt VI**
Georgia Institute of Technology

1. INTRODUCTION

This paper describes experimental results based on a relatively new technique for generating test data. This constraint based testing technique has been described in [DeMi88a] and [DeMi87a]. Here we describe a set of experiments that measure the technique. The method uses only the information provided by the source code to automatically generate test data that is guaranteed to be mutation-adequate.

Program mutation is a method of measuring test data quality. A test set is adequate if it distinguishes the program being tested from all incorrect programs represented by a set of possible errors. These error-laden programs are called the mutants of the program being tested. It is sometimes useful to record the extent to which a program has been so distinguished. This is carried out by "marking" mutants either dead, alive or equivalent. A mutation score of a test set is then the percentage of all mutants that are either dead or equivalent. A test set is mutation-adequate if its score is 100%. Mutation-adequate tests have been shown experimentally [Girg86a] and analytically [Budd82a] to be very high quality tests.

Unfortunately, generating mutation-adequate tests can be a very labor-intensive task. To generate these tests, a tester interacts with an interactive mutation system (the most recent of which is the Mothra Software Testing Environment [DeMi86a, Offu87a]) to examine remaining live mutants and design tests that kill them. Our strategy for developing test cases has been to represent the conditions under which those mutants will die as mathematical constraints on the definition of tests and to automatically generate sets of data that satisfy the constraints. Our ultimate goal is to completely automate the testing and measurement process so that when a correct program is submitted for test an adequate test set is produced as evidence of correctness without human intervention; and when an incorrect program is submitted to the test environment the tester is provided—again, without human intervention—a set of tests on which the program fails and a list of possible errors and their locations suitable for input to a debugging tool.

The work described here is part of the Mothra Software Test Environment Project [DeMi86a] being carried out jointly by Purdue University and the Georgia Institute of Technology. A prototype implementation of this test data generation technique has been integrated with Version 1.2 of the Mothra testing system. This automatic test data generator is called Godzilla. A set of experiments have been carried out to measure the quality of the test data generated by this method, to measure the performance of the technique and the tool, and to measure ways in which the technique could be improved. The paper begins with

* This work supported in part under Contract F30602-85-C-0255 through Rome Air Development Center.

** Current address: Department of Computer Science, Clemson University, Clemson SC 29634, ofut@clemson.edu.

an overview of mutation analysis followed by a description of the test data generation technique. We present the rationale for the test generation method in terms of mutation analysis, since the two methods share a common underlying theoretical basis. Following this, the set of experiments are described. They were conducted using the Godzilla test data generator. The results, implications and discussions for improvements are presented.

2. *OVERVIEW OF MUTATION*

Mutation analysis is a fault based testing method that is more completely described elsewhere [DeMi86a, DeMi78a, DeMi79a]. Mutation analysis measures the quality of a set of independently created test cases. In practice, a tester interacts with an automated *mutation system* to determine the adequacy of a test data set and to improve that test data. This is done by forcing the tester to generate tests for specific types of errors. These errors are represented by a set of simple syntactic changes to the test program that produce a set of *mutant* programs.

For example, function **MAX** is shown in Figure 1 with four embedded mutants (lines preceded by a "#"). The convention when displaying mutants of a program is to show the original program with the mutated statements inserted immediately below the original statements. Each of these mutated statements represents a different (mutant) program where the mutated statement replaces the statement immediately preceding it. The original version of **MAX** returns the larger of its two integer inputs:

```
   FUNCTION MAX (M,N)
   MAX = M
#    MAX = N
#    MAX = ABS (N)
   IF (N.GT.M) MAX = N
#    IF (N.LT.M) MAX = N
#    IF (N.GE.M) MAX = N
   RETURN
```

Figure 1. Function MAX

Mutants are designed to represent common errors[1] that a programmer might make. The goal of the tester during mutation analysis is to create test cases that distinguish each mutant program from the original by causing it to produce different output. When the output of a mutant program is different from that of the original program on the same input, that mutant is considered *dead* because the set of test cases is complete enough to find that error. For example, the inputs ($M = 2, N = 1$) would produce the output $MAX = 2$ on the original program. On the first mutant shown in Figure 1, the output would be $MAX = 1$. Since the output of the mutant is incorrect, this test case would kill that mutant. This would demonstrate that the original program did not contain the error indicated by the dead mutant.

Each mutant is executed individually on each test case; a dead mutant is not executed against subsequent test cases. A set of test cases that kills all of the mutants is *adequate* relative to that set of mutants. Thus, mutation analysis indicates how well the program has been tested though the adequacy score.

1. Mutation analysis is partially based on the empirical *competent programmer assumption*, which states that though programmers do create erroneous programs, the programs they create are close to being correct.

A program that has been successfully tested against an adequate test set is either correct or contains an error that has not been modeled by the mutants. The theoretical properties of adequacy have been discussed by [Weyu88a] and [Budd82a]. In practice, a program that has been successfully distinguished from its mutants has been very thoroughly tested [DeMi79a].

It is generally impossible to kill all the mutants defined on a program because some changes have no effect. The last mutant shown in **MAX** is an example of an *equivalent* mutant that will always produce the same output as the original program and can never be killed. These equivalent mutants usually represent optimizations or de-optimizations of the original program. In current mutation systems, equivalent mutants are recognized by human examination or by relatively primitive heuristics [Bald79a, Tana81a]. In fact, a complete solution to the equivalence problem is not possible. **The recognition of equivalent mutants and the actual creation of test cases are the two most human-intensive and therefore the most expensive actions within current mutation systems.** Reducing the amount of human interaction necessary to perform mutation is a major goal of this work. In this paper, we report experiments on a solution to the problem of the test data generation problem. This same technique can also be used for partial solutions to the equivalent mutant detection problem; discussed further in [Offu88a].

3. AUTOMATICALLY GENERATING MUTATION-ADEQUATE TEST DATA

Practical test data generation techniques choose a subset of the possible inputs according to some testing rationale. The approach taken in constraint based testing is to describe the conditions necessary to kill each mutation of a program with mathematical constraints. This section describes the characteristics of these conditions. The program BSEARCH is used as an example of the concepts. **BSEARCH** is shown in Figure 2 with one embedded mutant that will also be used for the examples. The mutant is on statement 13 and is a variable replacement mutant that replaces the variable *MID* by *HIGH*. This mutant models the situation where a programmer has mistakenly used the wrong variable name.

To discover how to select a test case that kills an individual mutant, we must first recall that the mutant is represented as a syntactic change to a particular statement. Since every other statement in the mutated program is exactly the same as in the original program, it is apparent that as a minimum we must execute the mutated statement. Assuming the mutated statement is reached, the test case must be such that the state of the original program after executing the original statement differs from the state of the mutant program after executing the mutated statement. Although this state difference is necessary, it is not sufficient to kill the mutant. For the mutant to die, the difference in state must propagate through to the end of the program.

For a test case T to kill a mutant M appearing on line S of a program P, T has to have three broad characteristics. Note that these characteristics make increasingly specific requirements on the test case:

1. Reachability—The statement that contains M is executed by T.
2. Necessity—The state of M immediately following execution of S is different from the state of P at the same point.
3. Sufficiency—The final state of M differs from that of P.

Note that the necessity condition is similar to Morell's *creation* condition and the sufficiency condition is similar to Morell's *propagation* condition [More84a, More88a]. Whereas Morell's creation condition describes program states in which mutants would alter the state of the program [More84a], the necessity condition describes a test case that will cause a state difference in one mutant of the program.

```
      LOGICAL FUNCTION BSEARCH (LIST, ELEM)
      INTEGER LIST (10), ELEM, LOW, HIGH, MID
1     LOW = 1
2     HIGH = 10
3  10 MID = (LOW + HIGH) / 2
4     IF (HIGH.LT.LOW) THEN
5       BSEARCH = .FALSE.
6       RETURN
7     ELSE
8       IF (ELEM.EQ.LIST(MID)) THEN
9         BSEARCH = .TRUE.
10        RETURN
11      ELSE
12        IF (ELEM.GT.LIST(MID)) THEN
13          LOW = MID + 1
#           LOW = HIGH + 1
14        ELSE
15          HIGH = MID - 1
16        ENDIF
17        GOTO 10
18      ENDIF
19    ENDIF
      RETURN
```

Figure 2. Mutant of BSEARCH

3.1 Reachability

Reachability refers to the fact that the test case must reach, or execute, the statement that contains the mutant we are trying to kill. To generate test cases that are guaranteed to reach a particular statement S, an expression is needed to describe a path to be taken through the program. This expression is called a ***path expression.*** A path expression for a statement S in a program P is an algebraic expression that describes a condition or conditions on a test case that is sufficient to guarantee that P will reach S. For the mutant of **BSEARCH**, for example, the test case must reach statement 13, which means the tests in statements 4 and 8 must evaluate to be false and the test in statement 12 must evaluate to be true. The path expression for this mutant would require that *HIGH* be greater than or equal to *LOW*, *ELEM* not be equal to *LIST*(*MID*), and *ELEM* be greater than *LIST*(*MID*).

3.2 Necessity

The necessity condition is the key behind the constraint based test data generation technique. The *state* of a program is described by the value of program variables and internal variables such as the program counter. In order to kill a mutant, a test case must create a state in the mutant program that differs from the state of the original program. Since the mutant program and the original program are identical except for the mutated statement, if the states of the two versions of the program are equal after execution of the mutated statement, then the mutant will not die.

The necessity condition is reminiscent of Howden's weak mutation [Howd82a]. In weak mutation, the states of the original program and the mutant program are compared over a "component" of the program. The component[2] is a piece of the program that contains the

mutated statement. If we view the portion of the program up to and including the mutated statement as a component, then the necessity condition forces the mutated component to produce a different output from the original component. This is exactly the requirement of weak mutation.

In the *BSEARCH* example, this means that the value of the variable *LOW* after the assignment on statement 13 in the mutant program must differ from the value of *LOW* in the original program. Since *LOW* is assigned the value of *MID*+1 in the original program and *HIGH*+1 in the mutant program, the necessity constraint for this mutant would require that *MID* and *HIGH* have different values (*MID* ≠ *HIGH*).

The necessary state difference can be described in terms of the same variables and program symbols that the mutant effects. Thus, this change depends solely on the mutant operators. An initial set of *necessity constraints* can be found described in [DeMi88a] and more completely in [Offu88a]. Constructing these necessity constraints from the mutants of the program assures that the test data will kill the mutants.

3.3 Sufficiency

Although a local state change is necessary to kill a mutant, it will not guarantee that the final state of the mutant program will differ from that of the original program. Once the states of the mutant program and the original program diverge, they may well converge to the same, final state. So the constraint is only sufficient to kill a mutant if it ensures that the final state of the mutant will differ from that of the original program.

Although other researchers are currently developing partial solutions to the sufficiency problem [Rich88a] the approach taken by this work is to assume that test cases that satisfy both the reachability and the necessity condition will also satisfy the sufficiency condition with very high probability. One of the experiments described is designed to approximate a measure of this probability.

4. GENERATING AND SOLVING CONSTRAINTS

In this section we describe how the constraints are generated, and more importantly, how they are solved in the Godzilla system. The Godzilla system is composed of five separate programs that total over 15,000 statements in the C programming language, so it is difficult to describe the complete system. In this section we present the more interesting highlights. Godzilla is more completely described in [Offu88a].

Necessity constraints are generated by a symbolic walk-through of the test program. This is actually the same algorithm that is used by Mothra's mutant maker [DeMi87b] except that rather than producing mutants, the walk-through produces constraints. Although this is the most important step in the test data generation process, it is technically the most simple part of the test data generation system. As each program symbol or expression is encountered, a set of rules are applied that defined the mutants that can be created from that symbol. These rules are used to form the necessity constraints.

For example, the mutant of BSEARCH in Figure 2 was created from the rule that states that each variable occurrence should be replaced by every other variable that is defined at the same scope. This is an example of a statement replacement mutant. The necessity constraint formed from this rule requires that the two variables should have unequal values. All the mutant operators that Mothra uses are described in [DeMi87b] and the formulation of

2. In his paper, Howden states that a component will "normally correspond to elementary computational structures in a program". Given a component C of a program P, there is a mutation transformation that produces C', and P' is the mutated version of P that contains C'.

necessity constraints from these operators is presented in [Offu88a].

The reachability condition is exactly the goal reflected by statement analysis. A lot of research has been carried out to find solutions to the statement analysis problem [Chus87a, Clar76a, Fran86a, Howd76a]. In general, finding a test case that is guaranteed to reach a statement is equivalent to the halting problem. In practice, data flow analysis or symbolic execution can usually suffice to derive an expression that will ensure that the statement will be executed.

Godzilla generates path expressions by performing a symbolic walk-through of the program [King76a, Howd77a, Howd78a]. Each statement is examined in turn and a current path expression is updated according to the type of statement. This path expression is initially TRUE, and at each control flow statement, the path expression is modified to reflect the potential flow of control. Also at each statement, the current path expression is added to the path expression list for that statement. The new path expression represents one way of reaching the statement. The current path expression is also updated by the expressions previously saved for that statement. Thus, the path expression will represent several ways of getting to the following statement.

Following this walk-through, the path expressions for each statement will represent all paths to the statement, not including loops. For the purpose of generating test cases to reach a statement, this should suffice, since the test case need only require that the statement is executed once.

The last step in the test case generation process is to generate values that satisfy the necessity and path expression constraints. Finding values to satisfy a set of constraints is a difficult problem that arises in such diverse areas as computability theory, operations research, and artificial intelligence. The approach taken by the Godzilla system is to employ a set of heuristics that work efficiently and produce satisfying test cases when the constraints have a simple form. These heuristics are described intuitively here, the full algorithm, with justification for why these heuristics do work for test case constraints, can be found in [Offu88a].

The satisfaction algorithm that Godzilla uses is a modification of the propagation algorithm [Gosl83a], which is in turn based on the topological sort algorithm. The propagation algorithm uses local information in the constraints to find values for variables, then uses back-substitution to simplify the remaining constraints. This simplification reduces expressions involved in the constraints through substitution and also reduces the domain of values that individual variables can take on. When this process comes to a halt (when no more simplification can be done), a heuristic is employed to choose a value for one variable in the constraints. This value is chosen arbitrarily from the current domain of values that the variable can take on. This value is then back-substituted into the remaining constraints and the process is repeated until all variables have been assigned a value.

This procedure is called the "domain reduction algorithm" because the domain of values that each variable can take on is successively reduced until an assignment is made. This is a brief, intuitive description of the procedure that describes the major points without the full details, which can be found in [Offu88a]. Domain reduction is a heuristic-based procedure that succeeds when constraints are simple and non-linear. Although it is guaranteed to terminate, it will not always generate a correct solution. Domain reduction employs randomness as part of the heuristics. This randomness is important because we sometimes wish to satisfy constraints multiple times in order to generate different test cases.

5. EXPERIMENTS WITH A CONSTRAINT BASED TEST DATA GENERATOR

The test data generator is a tool that uses the techniques described above to automatically generate test data for Mothra. This test data is designed specifically to kill the set of mutations that are defined on the program. The components of Godzilla are described in

detail in [DeMi89a]. In brief, the generator creates the path expression constraints, the necessity constraints, then combines the constraints and generates the test cases.

Various measurements of the test data generator have been taken to answer questions about the effectiveness of the technique, the quality of the solutions to the technical problems, and the quality of the implementation. Five studies are presented here. First the test cases are measured as a set, then on an individual basis in terms of how effective they are at finding errors. Then results of a comparison of this technique with other test data generation techniques in terms of effectiveness are presented. Then the time to create and execute the test data using the generator is compared to the time required by a human testing. Finally, a way of improving the performance of this technique is discussed and an estimate of how much improvement could be expected is undertaken.

A suite of five Fortran-77 programs were chosen for these studies. The five programs were taken from the literature and chosen to represent different types of problems to exercise the generation capabilities in as wide a manner as possible. The five programs are BUBBLE, DAYS, FIND, GCD, and TRITYP. The programs are listed in the appendix and described below.

BUBBLE is an array manipulation program that implements the bubble-sort algorithm to sort elements of a numeric array. DAYS is taken from Geller [Gell78a] and Budd's dissertation [Budd80a] and is a function that computes the number of days between two given dates. FIND was studied by Hoare [Hoar71a] and by DeMillo, Lipton and Sayward [DeMi78a] and accepts an array A of integers and an index F. It returns the array with every element to the left of $A(F)$ less than or equal to $A(F)$ and every element to the right of $A(F)$ greater than or equal to $A(F)$. GCD was presented by Bradley [Brad70a] and studied by Budd [Budd80a] and computes the greatest common divisor of the input array. TRITYP has been widely studied in software testing [Clar83a, DeMi78a, Acre79a, Rama76a] etc., so is perhaps the "canonical" software testing example. TRITYP takes three integers as input that represent the relative lengths of the sides of a triangle. TRITYP classifies the triangle is equilateral, isosceles, scalene or illegal.

These programs are referred to collectively as "the test program suite". These experiments were conducted using the Mothra mutation testing system, version 1.2, running on a Microvax II.

5.1 Adequacy of the Test Cases

Test case adequacy is the simplest and in some ways the most direct measurement of this test case generation method. The rationale behind the creation of the test cases is mutation-adequacy. Test cases are constructed precisely to score well on a mutation analysis system—to kill mutants. So the most direct measurement of performance is to calculate the *mutation score* of the test data. The mutation score (MS) is the ratio of dead over non-equivalent mutants. If the total number of mutants is M, the number of dead mutants is K, and the number of equivalent mutants is E, then the mutation score is calculated as:

$$MS = \frac{K}{(M - E)}.$$

The mutation score is a quantitative measure of not only how well the test data approximates adequacy, but also of how well the data tests the program [DeMi86a, Acre79a, Budd80b]. For this experiment, test cases were generated for each of the programs in the suite, all mutants were generated, and each test case was executed against all live mutants. This is the typical way that testers use Mothra. The results of this experiment are displayed in Table 1. The first column, TC, is the number of test cases generated, the next four columns (M, K, E, and MS) are from the mutation score formula, and the last column, *Time*, represents the length of wall clock time in minutes and seconds that Godzilla

	TCs	*M*	*K*	*E*	*MS*	*Time*
BUBBLE	32	339	304	35	1.00	0:22
DAYS	419	3016	2624	139	.95	7:02
FIND	58	1029	953	75	.99	2:27
GCD	325	5063	4747	298	.99	14:24
TRITYP	420	970	862	107	.99	10:53

TABLE 1. Test Case Adequacy Results

took to generate the test cases.

As can be seen, the test data scored 95 percent or more on each of the five programs. Practical experience has shown that it is extremely difficult to manually create test data that scores above 95 percent on a mutation system. In fact, one of the authors spent approximately 30 hours constructing a set of test data to kill the mutants for TRITYP. Given the test data that was produced automatically, it was the work of a few minutes to manually find test cases that killed TRITYP's remaining eight mutants.

5.2 Test Case Precision Experiment

The adequacy experiment measures the test data set as a whole. Each test case is constructed with a particular goal—to kill an individual mutant. For each test case to be effective, the test case must actually kill the specific mutant it was targeted for. Measuring the test cases individually eliminates the *overkill* effect, where some test cases not only kill their target mutants, but also kill mutants they were not targeted for and that may not have been killed by the test case that was targeted for them.

To measure the test cases individually, each test case is executed against only the mutant(s)[3] that it was designed to kill. This is called a "test case precision experiment" because it measures how precise each test case is at killing it's target mutants. The percentage of mutants killed gives an indication of how often this test data generation technique works.

If the number of mutants that were executed by some test case is M' and K is the total number of mutants that were killed, then the precision is defined as:

$$P = \frac{K}{M'}$$

Table 2 shows the precision of the test cases of the five programs in the program test suite. The *MS* column is the mutation score as calculated by the number of mutants killed over the total number of non-equivalent mutants. These scores are quite low—but also slightly misleading. Because Godzilla does not solve for constraints involving internal variables (and other special cases, such as deeply nested constraints), there were many mutants that did not have a test case constructed. That is, the system did not really try to kill these mutants.

3. When two mutants result in identical constraints, only one test case is generated, and this test case is targeted for both mutants. Ideas for expanding this notion to satisfying multiple but different constraints with the same test case are described later in this paper, under Combination of Test Case Constraints.

	TCs	*MS*	*P*
BUBBLE	32	.40	.94
DAYS	419	.18	.61
FIND	58	.09	.78
GCD	325	.14	.72
TRITYP	420	.50	.81

TABLE 2. Effectiveness Experiment Results

This table shows us that the overkill effect is quite important. Unfortunately, it also shows a wide variation in the precision over even a relatively small number of programs.

One way to use the overkill effect to our advantage is to satisfy the constraints multiple times to create multiple test cases per constraint. Although each additional test case implies additional cost for the testing process, we can anticipate that few test cases will be needed per constraint. If looked at probabilistically, each time we satisfy a set of constraints to create a test case, the test case has some probability (say ρ) of killing the mutant. If we satisfy the constraints multiple times we are making independent choices from among the test case set.[4] Thus, the probability of choosing at least one effective test case over n trials (Γ) is a function that grows very quickly with n:

$$\Gamma = 1 - (1-\rho)^n$$

This means that we can increase the effectiveness of our test case set by generating multiple test cases. Of course, adding test cases makes the testing process more expensive by increasing the overhead of managing the test cases and the time spent verifying that each test case creates correct output. This can be a welcome tradeoff, however, because as we increase the number of test cases (and the cost) we can also expect to increase the effectiveness of the test case set.

5.3 Adequacy Comparisons

Another important measurement of any test data generation technique is how the test data compares with the data generated by other techniques. Comparisons of this nature are surprisingly rare in the literature. There seem to be two difficulties with performing these experiments; one is the lack of automated tools that implement the known testing techniques and the other is the question of how to compare the testing techniques. Ideally, testing techniques should be compared using software developed and tested in a production environment [Basi87a]. The practical problems of accessing and testing such software seems to have limited this type of experimentation.

Measuring the test data adequacy is a way of comparing testing techniques that has been used in recent studies [Girg86a, DeMi81a, Ntaf84a]. Adequacy not only measures the error-detecting capabilities of the test data, but also gives an indication of how well the software will be tested using the test data [Budd80b]. Relative adequacy of a test data set can be easily measured using a mutation system such as Mothra.

An experiment that used an earlier mutation system to measure the adequacy of several test data generation techniques was presented by DeMillo et al [DeMi81a]. They used the

4. There are generally many test cases that will satisfy the constraints. The domain reduction algorithm incorporates randomness partially so that it can generate different test cases for the same constraints.

TRITYP program. Since no automated tools were available for the techniques, test data was generated by hand to satisfy the criteria of each of the five techniques studied. We repeated this experiment using the Mothra mutation system. To ensure repeatability, the same version of TRITYP that was used in the previous study was used. This version of TRITYP is slightly different from the version used in the other experiments presented here and is listed in the appendix as "Unstructured TRITYP".

The comparison experiment was extended to use test data generated with the Godzilla implementation of the constraint based testing technique. Table 3 gives the results of the test data executed against the mutants generated by the Mothra system.

Technique	TCs	K	MS
Statement Analysis	5	642	.667
Branch Analysis	9	749	.778
Specifications Analysis	13	780	.811
Minimized Domain Analysis	36	932	.969
Domain Analysis	75	943	.980
Constraint Based Testing	536	959	.997

TABLE 3. Summary of Comparison Results

As pointed out in the paper by DeMillo, Hocking and Merritt [DeMi81a], data sets with high mutation scores may be considered superior to those with low scores. Every mutant that is left alive by a test case set represents an error that the data could not detect. So a test set that earns a higher mutation score has demonstrated more error detection power.

As can be seen from the table, the current prototype version of constraint based testing is producing results that are better than the five techniques studied in this experiment. Of course, one could argue that in comparison with domain analysis (the most effective of the other five techniques), we added 461 test cases to kill 16 mutants, surely an expensive approach. However, an examination of the 536 constraint based test cases shows that only 46 were actually effective at killing mutants, compared with 41 effective test cases from the domain testing technique. Furthermore, since the constraint based test set was generated by an entirely automated method, the number of test cases should be balanced against the labor-intensiveness of the other techniques.

Though exact time statistics were not kept for the original experiment, estimates by the participants indicate that it took approximately seven man-weeks to hand-generate the data and execute the data against the mutants. The automated method took two hours and 20 minutes. Moreover, this time was almost entirely computer time—the tester simply pushed the "go" button and functioned as the oracle. This human versus machine time tradeoff was the subject of our next experiment.

5.4 Mutant Detection Time

The original goal of this research was to automate the process of generating test data within a mutation system. The most direct way to measure its performance is in terms of improvement over the previous method—which was manual entry of test data. To do this, test data was generated with the Godzilla system and executed by Mothra. The machine time, human time, and mutation score were recorded. Next, the author attempted to construct test data to reach the same mutation score, keeping track of the same times. Since one element of the measurement is the tester, wall clock time was used and the test cases were executed on an unloaded Microvax II.

The hand-generation portion of this experiment was done by the second author. This is a subject who is very familiar with mutation analysis as well as each of the programs and has also had much experience generating test cases for mutation analysis systems. This

experiment was performed for each of the five programs in the test data generation suite.

Table 4 presents the times to generate and execute test cases by hand and by Godzilla. All times are in wall clock minutes. The "Human Time" column gives the number of minutes used by the tester to create the test data, and the "Machine Time" is the number of minutes used by the computer to execute all test cases and, for Godzilla, to generate the test cases.

The "Oracle Time" is an estimate of the time spent verifying the output of the test cases on the original program. The tester served as the oracle and the time was measured using a stopwatch to time the examination of the output for 25 test cases per program to estimate an average time for verifying each test case. The average time spent verifying each test case was then multiplied by the number of test cases to get the total time used by the oracle. Though this is only a rough estimate, the fact that the same average time per test case was used for both the automatically generated and hand-generated test sets eliminates most of the inaccuracy. The point is that each additional test case requires more work from the human to serve as oracle.

	AUTHOR				GODZILLA			
	Number TCs	Human Time	Machine Time	Oracle Time	Number TCs	Human Time	Machine Time	Oracle Time
BUBBLE	9	9	2	1	32	1	3	2
FIND	25	1328	45	3	58	1	26	8
DAYS	52	503	99	26	419	1	286	210
TRITYP	56	609	45	14	420	1	95	105
GCD	87	6835	198	29	325	1	504	152

TABLE 4. Test Data Generation Times

As can be seen, the differences are dramatic. The time to set up to generate test data automatically is constant (indeed, practically negligible), whereas the human time spent analyzing the program's mutants and developing test data generally increased as the programs got larger and more complicated. The machine time used during automatic generation was around twice the time used during human generation. The time spent executing mutants was the dominating factor, rather than the time spent generating test cases.

The *speedup* (S) of generating the data automatically can be defined as the time used by human generation over the total time used by automatic generation. Table 5 shows the speedup from the data in Table 4:

	S
BUBBLE	2.00
DAYS	1.26
FIND	39.31
GCD	10.75
TRITYP	3.32

TABLE 5. Speedup of Automated Method

The values of the speedup for the programs ranged from 1.26 to 39.31. This is quite a variation but it is not surprising that less speedup was achieved when it took more time to verify correctness of the output. Although this study does not tell us conclusively how much human effort will be saved by using this technique to automatically generate test data, we can certainly conclude that constraint based testing does save significant time, effort and expense.

5.5 Combination of Test Case Constraints

The last experiment performed is aimed at reducing the number of test cases created by the automatic generator. In the experiments comparing a human tester with the automated generator, we saw that the generator produced from four to five times as many test cases as a human tester. Upon further analysis, many of these test cases did not actually distinguish any mutants because their target mutant(s) had been previously killed by other test cases. This implies some overlap in the effectiveness of the test case The experiment to measure this overlap attempts to "combine" the test case constraints. The idea is that many constraints describe overlapping sets of test cases, so that a single test case can be used to satisfy multiple constraints. To measure this, we attempt to satisfy multiple constraints with one test case, then measure the adequacy of the resulting set of test cases through mutation analysis.

An option to perform this combination during constraint satisfaction was implemented in Godzilla. All the constraints for a program are stored in a table and the constraint satisfier generates test cases for each constraint in turn—looping through the table. Godzilla was modified so that when a test case was generated, each subsequent constraint in the table was checked. If the test case also satisfied other constraints, those constraints were not satisfied later. This undoubtably will not generate the most optimal set of test cases, but it did significantly reduce the number of test cases required as well as being algorithmically simple and requiring no algebraic manipulation of the test cases.

	Full			Combine		
	TCs	Killed	MS	TCs	Killed	MS
BUBBLE	32	304	1.00	5	303	.99
DAYS	419	2624	.95	83	2607	.91
FIND	58	953	.99	32	938	.98
GCD	325	4747	.99	58	4636	.97
TRITYP	420	862	.99	37	794	.92

TABLE 6. Combining Constraints Results

Table 6 shows the results of this "multiple satisfaction" method of test case reduction. The first three columns repeat the previous results of generating test cases with no reduction. The next column gives the number of test cases generated by multiple satisfaction. The last two columns give the number of mutants the reduced set of test cases killed and the resulting mutation score.

The decrease in mutation score by the reduced set was not large (between one and seven percent). This means that it is possible to trade off a large amount of processing time for a small amount of testing strength.

To see how much improvement is achieved, we must compare the number of mutants killed as well as the number of test cases created. Let the number of test cases generated without constraint combination be T and the number generated with constraint combination be T'. The number of mutants killed by T is K and the number killed by T' is K'. The ratios of these numbers can be defined as:

$$\sigma = \frac{T'}{T}, \qquad \upsilon = \frac{K'}{K}$$

Our goal when reducing the number of constraints is to reduce the ratio of test cases generated (σ) without reducing the ratio of mutants killed (υ). In a (hypothetical) perfect case, we would reduce the number of test cases to one without effecting the number of mutants killed ($\sigma = 1/T$ and $\upsilon = 1$). In the worst case, we would reduce the number of

mutants killed to zero without lowering the number of test cases ($\sigma = 1$ and $\upsilon = 0$). We can define the *reduction efficiency* (R) as:

$$R = \frac{\upsilon}{\sigma} = \frac{K'}{K} * \frac{T}{T'}$$

R grows larger as the number of test cases decreases and grows smaller as the number of mutants killed decreases. As the number of test cases approaches one, R approaches infinity (perfection) and as the number of mutants killed approaches zero R approaches zero. Table 22 shows values for R calculated for the experiment.

	R
BUBBLE	6.4
DAYS	5.0
FIND	1.8
GCD	5.5
TRITYP	10.5

TABLE 7. Reduction Efficiency of Combining Constraints Experiments

The reduction efficiency was well above 1 for each of the five programs. If $R = 1$, then there would have been no improvement, so that can be regarded as a cutoff point. The fact that R varies so much indicates that we can expect very different amounts of improvement from combining constraints depending on the program being tested. Note that the number of mutants killed was not changed appreciably for any of the five programs—the variation was mainly due to the number of test cases created. TRITYP, whose reduction coefficient was above 10, had over 90% fewer test cases created, whereas FIND only had about 45% fewer test cases.

6. *CONCLUSIONS*

This work is the first attempt to generate test data based on program mutation. It solves a major problem with using mutation analysis as a practical method for testing software, that of creating test data. The method for generating test data has been fully implemented and integrated with Version I of the Mothra testing system—a mutation analysis based testing system for Fortran 77. The Godzilla implementation is composed of over 15,000 lines of *C* code that includes the ability to create constraints to kill mutants, perform symbolic executions of Fortran 77 programs to derive reachability constraints, and satisfy the constraints to generate test cases for the test program.

Because of its mutation analysis basis, the test data generated subsumes the error detection capabilities of such test case generation methods as branch analysis and domain analysis. This technique integrates the test data generation capabilities of path analysis techniques with the error detection capabilities of mutation analysis. The experiments that are described in this paper verify that the test data generation technique create test cases that score well on the mutation system. The technique is at least competitive with other test data techniques and may well be more powerful. Moreover, because the necessity constraints are derived from rules to describe test cases, the technique can be easily extended to handle other types of errors. These experiments also show that this method is a much more cost effective means of creating mutation-adequate test sets than by the current human methods.

REFERENCES

[Acre79a] Acree, A. T., T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation Analysis. TECHNICAL REPORT GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, September 1979.

[Bald79a] Baldwin, D. and F. Sayward. Heuristics for Determining Equivalence of Program Mutations. RESEARCH REPORT #276, Department of Computer Science, Yale University, 1979.

[Basi87a] Basili, V. R. and R. W. Selby. Comparing the Effectivness of Software Testing Strategies. *IEEE TRANSACTIONS OF SOFTWARE ENGINEERING* **13**, no. 12 (December 1987).

[Brad70a] Bradley, G. H. Algorithm and Bound for the Greatest Common Divisor of *n* Integers. *COMMUNICATIONS OF THE ACM* **13**, no. 7 (July 1970): 433-436.

[Budd80b] Budd, T., R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs. TECHNICAL REPORT GIT-ICS-80/01, Georgia Institute of Technology, 1980.

[Budd80a] Budd, T. A. Mutation Analysis of Program Test Data. PHD DISSERTATION, Yale University, New Haven CT, 1980.

[Budd82a] Budd, T. A. and D. Angluin. Two Notions of Correctness and Their Relation to Testing. *ACTA INFORMATICA* (Springer-Verlag) **18**, no. 1 (November 1982): 31-45.

[Chus87a] Chusho, T. Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing. *IEEE TRANSACTIONS OF SOFTWARE ENGINEERING* **13**, no. 5 (May 1987).

[Clar76a] Clarke, L. A. A System to Generate Test Data and Symbolically Execute Programs. *TRANSACTIONS ON SOFTWARE ENGINEERING* **2**, no. 3 (September 1976): 215-222.

[Clar83a] Clarke, L. A. and D. J. Richardson. The Application of Error-Sensitive Testing Strategies to Debugging. *SYMPOSIUM ON HIGH-LEVEL DEBUGGING* (ACM SIGSOFT/SIGPLAN) (March 1983): 45-52.

[DeMi78a] DeMillo, R. A., R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *COMPUTER* **11**, no. 4 (April 1978).

[DeMi79a] DeMillo, R. A., F. G. Sayward, and R. J. Lipton. PROGRAM MUTATION: A New Approach to Program Testing. *INFOTECH INTERNATIONAL STATE OF THE ART REPORT: PROGRAM TESTING* (Infotech International) (1979).

[DeMi81a] DeMillo, R. A., D. E. Hocking, and M. J. Merritt. A Comparison of Some Reliable Test Data Generation Procedures. TECHNICAL REPORT GIT-ICS-81/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, April 1981.

[DeMi86a] DeMillo, R. A. and E. H. Spafford. The Mothra Software Testing Environment. *PROCEEDINGS OF THE 11TH NASA SOFTWARE ENGINEERING LABORATORY WORKSHOP*, Goddard Space Center (December 1986).

[DeMi87b] DeMillo, R. A., D. S. Guindi, K. N. King, E. W. Krauser, W. M. McCracken, A. J. Offutt, and E. H. Spafford. Mothra Internal Documentation, Version 1.0. TECHNICAL REPORT GIT-SERC-87/10, Software Engineering Research Center, Georgia Institute of Technology, Atlanta GA, 1987.

[DeMi87a] DeMillo, R. A. and A. J. Offutt. Constraint Based Automatic Test Data Generation. TECHNICAL REPORT, GIT-SERC-87/17, Software Engineering Research Center, Georgia Institute of Technology, Atlanta GA, 1987.

[DeMi88a] DeMillo, R. A., D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An Extended Overview of the MOTHRA Mutation System. *PROCEEDINGS OF THE SECOND WORKSHOP ON SOFTWARE TESTING, VERIFICATION AND ANALYSIS*, Banff Alberta (July 1988).

[DeMi89a] DeMillo, R. A., E. W. Krauser, A. J. Offutt, R. J. Martin, and E. H. Spafford. The MOTHRA Tool Set. *PROCEEDINGS OF THE HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES*, Kailua-Kona HI (January 1989). (To be presented.)

[Fran86a] Frankl, P. G. and E. J. Weyuker. Data Flow Testing in the Presence of Unexecutable Paths. *PROCEEDINGS OF THE WORKSHOP ON SOFTWARE TESTING CONFERENCE* (IEEE Computer Society Press) (July 1986): 4-13.

[Gell78a] Geller, M. Test Data as an Aid in Proving Program Correctness. *COMMUNICATIONS OF THE ACM* **21**, no. 5 (May 1978).

[Girg86a] Girgis, M. R. and M. R. Woodward. An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria. *PROCEEDINGS OF THE WORKSHOP ON SOFTWARE TESTING CONFERENCE* (IEEE Computer Society Press) (July 1986): 64-73.

[Gosl83a] Gosling, J. Algebraic Constraints. PHD DISSERTATION, Department of Computer Science, Carnegie-Mellon University, Pittsburgh PA, 1983.

[Hoar71a] Hoare, C. A. R. Proof of a Program: FIND. *COMMUNICATIONS OF THE ACM* **14**, no. 1 (January 1971).

[Howd76a] Howden, W. E. Reliability of the Path Analysis Testing Strategy. *TRANSACTIONS OF SOFTWARE ENGINEERING* (IEEE) **2**, no. 3 (September 1976).

[Howd77a] Howden, W. E. Symbolic testing and the DISSECT Symbolic Evaluation System. *TRANSACTIONS OF SOFTWARE ENGINEERING* (IEEE) **SE-3**, no. 4 (July 1977).

[Howd78a] Howden, W. E. An Evaluation of the Effectiveness of Symbolic Testing. *SOFTWARE PRACTICE AND EXPERIENCE* **8** (1978): 381-397.

[Howd82a] Howden, W. E. Weak Mutation Testing and Completeness of Test Sets. *TRANSACTIONS ON SOFTWARE ENGINEERING* (IEEE) **8**, no. 2 (July 1982): 371-379.

[King76a] King, J. C. Symbolic Execution and Program Testing. *COMMUNICATIONS OF THE ACM* **19**, no. 7 (July 1976): 385-394.

[More84a] Morell, L. J. A Theory of Error-Based Testing. TECHNICAL REPORT TR-1395, PHD DISSERTATION, Department of Computer Science, University of Maryland, College Park MD, 1984.

[More88a] Morell, L. J. Theoretical Insights into Fault-Based Testing. *PROCEEDINGS OF THE SECOND WORKSHOP ON SOFTWARE TESTING, VERIFICATION AND ANALYSIS*, Banff Alberta (July 1988).

[Ntaf84a] Ntafos, S. C. An Evaluation of Required Element Testing Strategies. *PROCEEDINGS OF THE SEVENTH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING* (IEEE Computer Society) (March 1984): 250-256.

[Offu87a] Offutt, A. J. and K. N. King. A Fortran 77 Interpreter for Mutation Analysis. *1987 SYMPOSIUM ON INTERPRETERS AND INTERPRETIVE TECHNIQUES* (ACM SIGPLAN), St. Paul MN (June 1987): 177-188.

[Offu88a] Offutt, A. J. Automatic Test Data Generation. GIT-ICS 88/28, PHD DISSERTATION, Georgia Institute of Technology, Atlanta GA, 1988.

[Rama76a] Ramamoorthy, C. V., S. F. Ho, and W. T. Chen. On the Automated Generation of Program Test Data. *TRANSACTIONS ON SOFTWARE ENGINEERING* (IEEE) 2, no. 4 (December 1976).

[Rich88a] Richardson, D. J. and M. C. Thompson. The RELAY Model for Error Detection and its Application. *PROCEEDINGS OF THE SECOND WORKSHOP ON SOFTWARE TESTING, VERIFICATION AND ANALYSIS*, Banff Alberta (July 1988).

[Tana81a] Tanaka, A. Equivalence Testing for Fortran Mutation System Using Data Flow Analysis. MS THESIS, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, 1981.

[Weyu88a] Weyuker, E. The Evaluation of Program-Based Software Test Data Adequacy Criteria. *COMMUNICATIONS OF THE ACM* **31**, no. 6 (June 1988): 676-686.

APPENDIX

Fortran 77 Program Listings

This appendix contains listings of six Fortran-77 programs. The first five are BUBBLE, DAYS, FIND, GCD, and TRITYP. The last program is called Unstructured TRITYP and was discussed under "Adequacy Comparisons". It is an older version of TRITYP that is used to repeat an earlier experiment. For each program, some indication of its source is given in comments.

BUBBLE Program

```
      SUBROUTINE BUBBLE (A)
      INTEGER A (5)

C     Sort A using the bubble sort technique.
C     Written by Jeff Offutt, 1986.

      INTEGER N, ITMP

      N = 5
      DO 200 J = N-1, 1, -1
        DO 100 I = 1, J, 1
          IF (A (I).LE.A (I+ 1)) GOTO 100
            ITMP = A (I)
            A (I) = A (I+ 1)
            A (I+ 1) = ITMP
100     CONTINUE
200   CONTINUE
      RETURN
      END
```

DAYS Program

```
      INTEGER FUNCTION DAYS (DAY1, MONTH1, DAY2, MONTH2, YEAR)
      INTEGER DAY1, MONTH1, DAY2, MONTH2, YEAR

C     Calculate number of DAYS between the two given days.
C     DAY1 and DAY2 must be in same year.
C     Taken from Budd's thesis, pg 65, repeated from Geller [Gell78].
C     Translated from COBOL by Jeff Offutt, 3/88.

      INTEGER DAYSIN (12), I
      INTEGER M4, M100, M400

C     Start range from 1-10000 for year.
      ASSERT (DAY1.LE.31.AND.DAY2.LE.31.AND.MONTH1.LE.12.AND.MONTH2.LE.12)
C     If the dates are in the same month, we can
C     compute the numer of days between them immediately.
      IF (MONTH2.EQ.MONTH1) THEN
        DAYS = DAY2 - DAY1
      ELSE
        DAYSIN (1) = 31
C       Are we in a leap year?
        M4  = MOD (YEAR, 4)
        M100 = MOD (YEAR, 100)
        M400 = MOD (YEAR, 400)
        IF ((M4.NE.0).OR.((M100.EQ.0).AND.(M400.NE.0))) THEN
          DAYSIN (2) = 28
        ELSE
          DAYSIN (2) = 29
        ENDIF
        DAYSIN (3)  = 31
        DAYSIN (4)  = 30
        DAYSIN (5)  = 31
        DAYSIN (6)  = 30
        DAYSIN (7)  = 31
        DAYSIN (8)  = 31
        DAYSIN (9)  = 30
        DAYSIN (10) = 31
        DAYSIN (11) = 30
        DAYSIN (12) = 31
C       Start with days in the two months.
        DAYS = DAY2 + (DAYSIN (MONTH1) - DAY1)
C       Add the days in the intervening months
        DO 10 I = MONTH1+1, MONTH2-1, 1
          DAYS = DAYSIN (I) + DAYS
10      CONTINUE
      ENDIF
      RETURN
      END
```

FIND Program

```
      SUBROUTINE FIND (A, N, F)
      INTEGER A (10), N, F

C     F is index into A(). After execution, all elements to the left of
C     A(F) are less than or equal to A(F) and all elements to the right of
C     A(F) are greater than or equal to A(F).
C     From DeMillo, Lipton, and Sayward [DeMi78], repeated from Hoare's
C     paper [Hoar70].

      INTEGER M, NS, R, I, J, W

      ASSERT (F.GE.1.AND.F.LE.N)
      M = 1
      NS = N
10    IF (M.GE.NS) GOTO 1000
      R = A (F)
      I = M
      J = NS
20    IF (I.GT.J) GOTO 60
30    IF (A(I).GE.R) GOTO 40
      I = I + 1
      GOTO 30
40    IF (R.GE.A(J)) GOTO 50
      J = J - 1
      GOTO 40
50    IF (I.GT.J) GOTO 20

      W = A (I)
      A (I) = A (J)
      A (J) = W
      I = I + 1
      J = J - 1
      GOTO 20

60    IF (F.GT.J) GOTO 70
      NS = J
      GOTO 10
70    IF (I.GT.F) GOTO 1000
      M = I
      GOTO 10
1000  RETURN
      END
```

```
      SUBROUTINE GCD (N, A, Z, IGCD)
      DIMENSION A(10), Z(10)
      INTEGER A, Z, N

C     Calculate greatest common divisor of elements of A.
C     N gives length of A(), A() is destroyed.
C     Z() is output array of N multipliers.
C     Used in Budd's thesis, repeated from [Brad70a].

      INTEGER C1, C2, Y1, Y2, Q

C     In: N,A
C     Out: Z, IGCD
      ASSERT (N.LE.10.AND.N.GE.1)

C     Find first non-zero integer.
      DO 1 M = 1, N
        IF (A(M).NE.0) GOTO 3
1     Z(M) = 0
C     All zero input results in zero GCD and all zero multipliers.
      IGCD = 0
      RETURN

C     If last number is the only non-zero number, exit immediately.
3     IF (M.NE.N) GOTO 4
      IGCD = A(M)
      Z(M) = 1
      RETURN

4     MP1 = M+1
      MP2 = M+2
C     Check the sign of A(M).
      ISIGN = 0
      IF (A(M).GE.0) GOTO 5
      ISIGN = 1
      A(M) = -A(M)
C     Calculate GCD via N-1 applications of the GCD algorithm for two
C     integers. Save the multipliers.
5     C1 = A(M)
      DO 30 I = MP1, N
        IF (A(I).NE.0) GOTO 7
        A(I) = 1
        Z(I) = 0
        GOTO 25
7       Y1 = 1
        Y2 = 0
        C2 = IABS (A(I))
10      Q = C2 / C1
        C2 = C2 - Q*C1
C       Testing before computing Y2 and before computing Y1 below
C       saves N-1 additions and N-1 multiplications.
        IF (C2.EQ.0) GOTO 20
        Y2 = Y2 - Q*Y1
        Q = C1 / C2
```

```
        C1 = C1 - Q*C2
        IF (C1.EQ.0) GOTO 15
        Y1 = Y1 - Q*Y2
        GOTO 10
15      C1 = C2
        Y1 = Y2
20      Z(I) = (C1 - Y1 * A(M)) / A(I)
        A(I) = Y1
        A(M) = C1
C       Terminate GCD calculations if GCD equals one.
25      IF (C1.EQ.1) GOTO 60
30    CONTINUE

40    IGCD = A(M)
C     Calculate multipliers.
      DO 50 J = MP2,1,-1
        K = I - J + 2
        KK = K + 1
        Z(K) = Z(K) * A(KK)
50      A(K) = A(K) * A(KK)
      Z(M) = A(MP1)
      IF (ISIGN.EQ.0) GOTO 100
      Z(M) = -Z(M)
100   RETURN

C     GCD found, set remainder of the multipliers equal to zero.
60    IP1 = I + 1
      DO 65 J = IP1, N
65      Z(J) = 0
      GOTO 40
      END
```

```
      FUNCTION TRIANG (I,J,K)
      INTEGER I,J,K

C     TRIANG is output from the routine:
C       TRIANG = 1 If triangle is scalene
C       TRIANG = 2 If triangle is isosceles
C       TRIANG = 3 If triangle is equilateral
C       TRIANG = 4 If not a triangle
C     From Ramamoorthy, Ho, and Chen [Rama76] and elsewhere.

C     After a quick confirmation that it's a legal
C     triangle, detect any sides of equal length
      IF (I.LE.0.OR.J.LE.0.OR.K.LE.0) THEN
        TRIANG=4
        RETURN
      ENDIF
      TRIANG=0
      IF (I.EQ.J) TRIANG=TRIANG+1
      IF (I.EQ.K) TRIANG=TRIANG+2
      IF (J.EQ.K) TRIANG=TRIANG+3
      IF (TRIANG.EQ.0) THEN

C       Confirm it's a legal triangle before declaring
C       it to be scalene
        IF (I+J.LE.K.OR.J+K.LE.I.OR.I+K.LE.J) THEN
          TRIANG = 4
        ELSE
          TRIANG = 1
        ENDIF
        RETURN
      ENDIF
C     Confirm it's a legal triangle before declaring
C     it to be isosceles or equilateral

      IF (TRIANG.GT.3) THEN
        TRIANG = 3
      ELSE IF (TRIANG.EQ.1.AND.I+J.GE.K) THEN
        TRIANG = 2
      ELSE IF (TRIANG.EQ.2.AND.I+K.GE.J) THEN
        TRIANG = 2
      ELSE IF (TRIANG.EQ.3.AND.J+K.GE.I) THEN
        TRIANG = 2
      ELSE
        TRIANG = 4
      ENDIF

      END
```

Unstructured TRITYP Program

```
      SUBROUTINE TRITYP (I,J,K, CODE)

      INTEGER I,J,K, CODE
      INTEGER MATCH
C     CODE is output from the routine:
C       CODE = 1 If triangle is equilateral
C       CODE = 2 If triangle is isosceles
C       CODE = 3 If triangle is scalene
C       CODE = 4 If not a triangle
C     From Ramamoorthy, Ho, and Chen [Rama76] and elsewhere.

C     Count matching sides
      MATCH = 0
      IF (I.EQ.J) MATCH = MATCH + 100
      IF (I.EQ.K) MATCH = MATCH + 200
      IF (J.EQ.K) MATCH = MATCH + 300
C     Select Possible scalene triangles
      IF (MATCH.EQ.0) GOTO 10
C     Select Possible isosceles triangles
      IF (MATCH.EQ.100) GOTO 20
      IF (MATCH.EQ.200) GOTO 30
      IF (MATCH.EQ.300) GOTO 40
C     Triangle must be equilateral.
      CODE = 1
      RETURN

C     Possible scalene
1     IF ((I+J).LE.K.OR.(J+K).LE.I.OR.(I+K).LE.J) GOTO 50
      CODE = 3
      RETURN

20    IF ((I+J).LE.K) GOTO 50
      GOTO 60
30    IF ((I+K).LE.J) GOTO 50
      GOTO 60
40    IF ((J+K).LE.I) GOTO 50
      GOTO 60
C     No triangle possible.
50    CODE = 4
      RETURN

C     Isosceles
60    CODE = 2
      RETURN
      END
```

Using Data Flow Analysis for Regression Testing

Thomas J. Ostrand
Software Technology Department
Siemens Research and Technology Laboratories
105 College Road East
Princeton, New Jersey 08540

tjo@cadillac.siemens.com
(609)-734-6569

Elaine J. Weyuker[1]
Courant Institute
of Mathematical Sciences
New York University
251 Mercer Street
New York, New York 10012

weyuker@csd2.nyu.edu
(212)-998-3082

Abstract

Dataflow analysis provides a way to select effective regression tests, based directly on the changes made to a program. If the program's original test cases were developed through data flow analysis, then the changes in data flow in the modified program can be used to guide the selection of a subset of the original tests as regression tests. It is also possible to derive new data flow-based tests that are specifically directed at finding errors introduced by the modifications.

We describe enhancements that will be made to the ASSET data flow testing tool to support the regression testing methods described in this paper. The enhanced tool will save relevant data flow information when a program is originally tested, and will analyze modified code to determine appropriate regression tests.

Thomas J. Ostrand is a Senior Research Scientist at Siemens Research Laboratories, where he works on methods and environments to support software testing.

Elaine J. Weyuker is an Associate Professor of Computer Science at the Courant Institute of Mathematical Sciences, New York University. Her research interests include software testing, software metrics, and software engineering.

Sixth Annual Pacific Northwest Software Quality Conference
Portland, Oregon
September 19-20, 1988

[1]Prof. Weyuker's research was supported in part by the National Science Foundation under Grant CCR-85-01614, and by the Office of Naval Research under Contract N00014-85-K-0414

Using Data Flow Analysis for Regression Testing

Thomas J. Ostrand
Elaine J. Weyuker

1 Introduction

Previously tested and accepted software is changed for many reasons, including specification changes, planned enhancements, tuning, adaptation to a new environment, and debugging. Studies estimate that from forty to seventy percent of the total life cycle effort for a software system is spent on these maintenance activities, and that the probability of introducing an error while making a change is between fifty and eighty percent (See Figure 1, taken from [Het84]). Given such statistics, testing software after it has been modified is at least as important as testing freshly produced code.

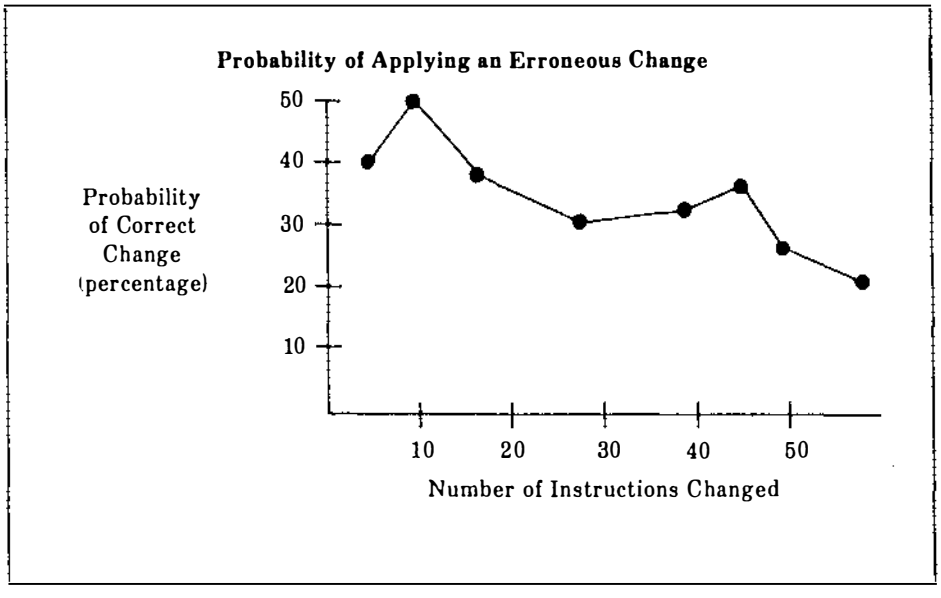


Figure 1: Errors Introduced When Code is Modified

Regression testing is the process of retesting changed software to determine whether faults were introduced as a result of the modifications. Regression testing is frequently done at the functional or system testing level, with the goal being to assure that existing functionality has not been damaged. We discuss here both this traditional type of regression testing, and also techniques for retesting individual modules that have been modified.

In the ideal case, whenever a program change has been made, all existing test cases would be rerun in order to determine whether the change introduced faults into the code. Using already devised test cases minimizes the effort needed to create the regression tests, and

also allows direct comparison of the output from the modified software against the original test results. Of course, this requires that the previous test cases and their results are stored.

However, rerunning all test cases whenever a program is changed is usually too expensive, and so a subset of the tests originally run is selected to check out the modifications. In this paper, we discuss intelligent ways of selecting this subset. In addition, existing test cases are frequently not sufficient for evaluating revised software, particularly if new functionality has been added or if the software's structure is changed. Since it is also usually too expensive to treat the revised code as though it were completely untested software and develop an entirely new set of test data, in these cases the regression tester needs some means of choosing additional tests.

Data flow analysis is a relatively new method for selecting and evaluating test data [RW82,RW85]. It has been applied both to unit and integration testing. In this paper we show how data flow analysis can be used both to select regression tests from previously defined test cases, and to generate additional tests for revised code. Data flow relationships within a single procedure are used to select regression test cases for unit testing of the procedure. Data flow relationships across procedure boundaries are used to develop regression tests for procedure calls.

In the next two sections we present an overview of data flow-based testing and of ASSET, an interactive tool for applying the method to programs. We then discuss three different bases for applying data flow analysis to regression testing.

2 Data Flow-Based Testing

For the last several years, we have been developing, refining, and experimenting with a theory of test data selection and evaluation based on the use of data flow information in the subject program [RW82,RW85,Wey84,FWW85,FW85,FW86,FW88]. In these papers we define and analyze a hierarchy of increasingly stricter criteria for selecting test cases. Theories and criteria with a similar underlying philosophy were developed by Herman [Her76], Ntafos [Nta84], and Laski and Korel [LK83], and a comprehensive comparison of the criteria was presented by Clarke et.al. in [CPRZ85]

These data flow testing criteria require the selection of test data that exercise certain paths from a point in a program where a variable is given a value or defined, to points at which that variable definition is subsequently used. By varying the required combinations of definitions and uses, a family of test data selection and adequacy criteria is defined.

The computational statements of a program (such as assignments, procedure calls, and input/output) can be uniquely decomposed into a set of disjoint *blocks* of statements. A block is a maximal sequence of simple statements, with the properties that it can only

be entered through the first statement, and whenever the first statement is executed, the remaining statements of the block are executed in the given order. A block ends either with a conditional transfer statement, or with a transfer of control to a statement that is in another block. All the statements of a program can be represented as a *flow graph*, which is a graph whose nodes correspond to the blocks of the program, and whose edges contain the predicates of the program's control statements (if..then..else, while, goto, etc.). Figure 2 shows the text of a simple program, and its corresponding flow graph.

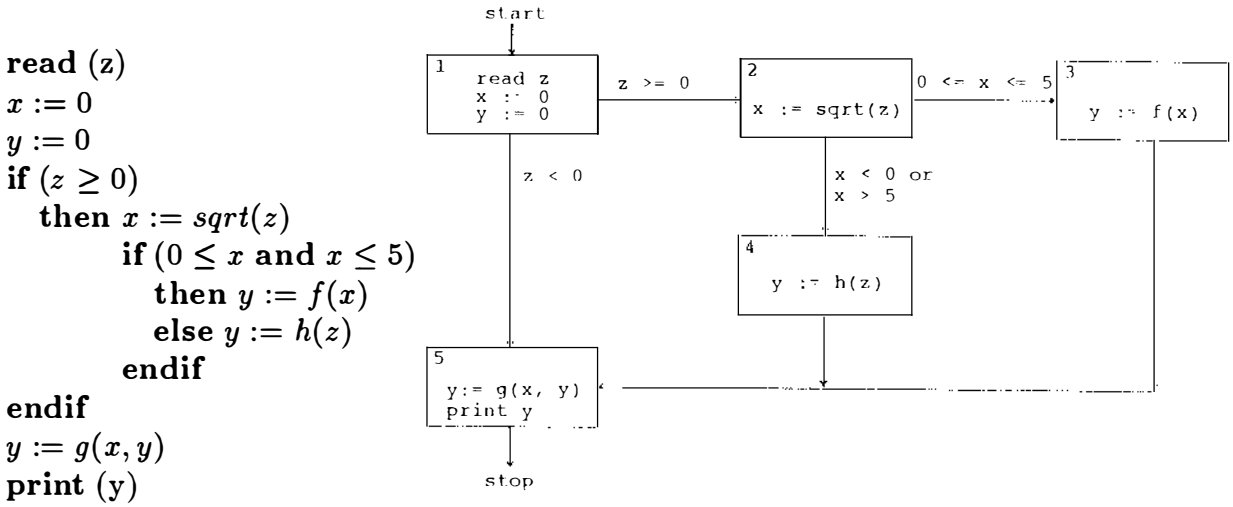


Figure 2: Sample Program

A *definition* of a variable in a program is an occurrence of the variable where it is given a new value, as in an input statement, the left-hand side of an assignment statement, or as an output parameter of a procedure call. A *use* of a variable is an occurrence where the variable does not receive a new value. We distinguish between two distinct types of uses: predicate uses (p-uses) and computation uses (c-uses). *P-uses* are uses in the predicate portion of a decision statement such as while...do, if...then...else, or repeat...until statements. *C-uses* are uses which are not p-uses, including variable occurrences in the right-hand side of an assignment statement, in an output statement, or use as an actual parameter in a procedure call. In the flow graph, c-uses always occur in the nodes, while p-uses always occur on the edges.

In Figure 2, node 1 contains definitions of the variables z, x, and y, node 5 has a definition of y, a c-use of x, and two c-uses of y, and the edge (1,5) has a p-use of z. (Assume that the two parameters of procedure g are only input parameters.)

A *definition-use association* is a triple (d, u, x), where x is a variable, d is a node containing a definition of x, u is either a node or an edge containing a use of x, and there is a subpath in the flow graph from d to u with no other definition of x between d and u. This definition captures the concept that the value of x that is used in u is the same

value that was assigned to x in d.

The data flow criteria require that test data be included which cause the traversal of subpaths from a variable definition to either some or all of the p-uses, c-uses, or a combination of p-uses and c-uses of that variable. For example, the *all-definitions* criterion requires sufficient test data such that every definition is used at least once. Put another way, this criterion requires that test data be included which cause the traversal of at least one subpath from each variable definition to *some* p-use *or some* c-use of that variable definition. The *all-c-uses* criterion requires that test data be included which cause the traversal of at least one path from each variable definition to *every* c-use of that variable definition. The *all-p-uses* criterion requires that test data be included which cause the traversal of at least one path from each variable definition to *every* p-use of that variable definition. The *all-uses* criterion requires that test data be included which cause the traversal of at least one subpath from each variable definition to *every* p-use *and every* c-use of that variable definition. The *all-du-paths* criterion requires that test data be included which cause the traversal of *every simple* subpath[2] from each variable definition to *every* p-use *and every* c-use of that variable definition. The relationships between these criteria are explored in [RW82] and [RW85]. Precise definitions of the entire family of criteria are included in [RW85] for a very simple language, and in [FW88] for Pascal.

3 The ASSET Test Evaluation Tool

ASSET is an interactive tool that enables a user to apply the data flow criteria to the testing of a program [FWW85,FW85]. When used for initial test evaluation, ASSET reports whether a given set of test cases for a program satisfies a particular data flow criterion specified by the user. ASSET analyzes the subject program's block structure, constructs its flow graph, and constructs an instrumented version of the program. The user then selects a data flow criterion, and specifies a set of test data for the program. ASSET then runs the program on the specified tests, collects the execution traces produced by each executed test case, and reports to the user which definition-use associations have not yet been exercised.

We are at present enhancing ASSET to support the regression testing methods that are described in this paper. The new version of the tool will save information during the original testing of a program, will analyze differences between modified code and the original program, and will aid the user in choosing sets of regression tests that meet certain criteria.

When data flow-based regression testing is to be performed, the user can request that ASSET record and maintain information relating test cases and paths for later use.

[2] A *simple subpath* is a subpath on which all nodes, with the possible exception of the first and last, are distinct.

This information is stored in three matrices, a *Def-Use* matrix, a *Node* matrix, and an *Edge* matrix. The rows of each matrix represent individual test cases. In the Def-Use matrix, the columns represent definition-use associations in the program. An entry in the Def-Use matrix can have five possible values; each of the values occurs in Figures 3 and 4 in reference to the program in Figure 2.

A **DUA** in position *(i,j)* indicates that the path traversed by test case i exercises definition-use association j. (Recall that a definition-use association consists of a triple (d,u,x), where d is a definition node, and u is either a node or an edge where the definition is used.) A **D** in position *(i,j)* indicates that the path traversed by test case i does not exercise definition-use association j, but does pass through the definition node. Similarly, a **U** in position *(i,j)* indicates that the path traversed by test case i does not exercise definition-use association j, but does pass through the use of the variable. A **DU** in position *(i,j)* indicates that the path traversed by test case i does not exercise definition-use association j, but it does pass through both the definition and use of the variable. This situation occurs when the variable is redefined in a node on the path that is between the definition and the use. A **0** in position (i,j) indicates that neither the definition nor the use node of association j is on the path traversed by test case i.

The Node matrix is a bit matrix whose columns represent the nodes of the flow graph. Element (i,j) is 1 if the path exercised by test case i includes node j, and 0 otherwise. The Edge matrix is also a bit matrix; its columns represent the edges of the flow graph. Element (i,j) is 1 if the path exercised by test case i includes the edge j, and 0 otherwise.

Figure 3 shows the c-use columns of the Def-Use matrix for the two test cases $z = 49$ and $z = -1$, applied to the program of Figure 2. Figure 4 shows the p-use columns. The third entry in the first row of Figure 4, for example, means that the input $z = 49$ traverses some subpath from node 2 to edge (2,4), and that the variable x is not redefined on that subpath.

D-U Assns	(1,2,z)	(1,4,z)	(1,5,x)	(1,5,y)	(2,3,x)	(2,5,x)	(4,5,y)	(3,5,y)
Test Cases								
z=49	DUA	DUA	DU	DU	D	DUA	DUA	U
z=-1	D	D	DUA	DUA	0	U	U	U

Figure 3: C-use columns of Def-Use matrix for test cases $z = 49$ and $z = -1$

D-U Assns	(1,(1,2),z)	(1,(1,5),z)	(2,(2,4),x)	(2,(2,3),x)
Test Cases				
z=49	DUA	D	DUA	D
z=-1	D	DUA	0	0

Figure 4: P-use columns of Def-Use matrix for test cases $z = 49$ and $z = -1$

The c-use associations of the program in Figure 2 are $(1,2,z)$, $(1,4,z)$, $(1,5,x)$, $(1,5,y)$, $(2,3,x)$, $(2,5,x)$, $(4,5,y)$, and $(3,5,y)$. The p-use associations are $(1,(1,2),z)$,

$(1,(1,5),z)$, $(2,(2,4),x)$, and $(2,(2,3),x)$. Consider the test case $z = 49$; this input traverses the path $(1,2,4,5)$, and exercises the definition-use associations $(1,2,z)$, $(1,4,z)$, $(2,5,x)$, $(4,5,y)$, $(1,(1,2),z)$, and $(2,(2,4),x)$. None of the other definition-use associations are exercised by this test case. Although both nodes 1 and 5 are executed on the path $(1,2,4,5)$, neither $(1,5,x)$ nor $(1,5,y)$ is satisfied, since both variables are redefined before node 5 is executed. If the test case $z = -1$ is selected, the path $(1,5)$ is traversed, and these two associations are satisfied.

4 The Data Flow Approach to Regression Testing

The most common approach for selecting regression tests is to use a (usually small) subset of the tests originally written for the software, primarily tests that verify the most important parts of the program's functions. This satisfies three major objectives of regression testing:

- the tests check whether the software's main functionality has deteriorated as a result of the modifications,
- the cost of producing the tests is minimized, since the tests themselves are already in existence,
- the test results can be compared directly with the results of the original test runs.

Such a set of "permanent" regression tests is intended to detect errors in the essential parts of the system's functionality whose code may or may not have changed. In addition, new test cases are needed when the system's functionality has changed and when the structure of the software has changed significantly. Data flow analysis provides means to select tests for all these situations. While the theory of data flow testing is that executing *all* definition-use associations of a particular type is an effective way to expose existing errors, the theory of data flow *regression* testing is that executing *changed* data flow associations is an effective way to expose errors in modified code. At the same time, portions of the program which do not interact in terms of data flow with the modified code are unlikely to be affected by the changes, and it is therefore reasonable to omit test cases that do not exercise the changed portions of the program.

The information collected by ASSET when it processes a source program provides the basis for choosing as regression tests two subsets of the original test set T. The first subset includes just those test cases that directly exercise the definition-use associations that have changed as a result of modifications to the program. This subset can be derived quickly from the Def-Use matrix, and is called the *Def-Use Set.* The second subset contains the first; it includes the test cases that exercise all paths that include any parts of the program that have changed. We call this second set of test cases the

Path Set. The Node and Edge matrices provide the information needed to quickly derive the Path Set.

The Def-Use set provides a minimal, yet effective, set of regression tests, since it includes precisely the tests that were created to exercise data flow relations between the specific nodes and edges of the original program that have been changed in the modified version. The Path Set is a larger set of tests that could be used for regression testing if a higher level of confidence is required, or if sufficient time and personnel are available.

In the following descriptions, we assume that the set of tests T used for the original program has been judged adequate with respect to one of the data flow criteria described in Section 2, that T has been stored for reuse, and that the matrices defined in Section 3 have been constructed. We want to test modified code either with existing test cases or with new cases that exercise definition-use associations that differ from those in the original code. The next three subsections describe three ways of selecting regression tests based on data flow:

- choosing a subset of previously executed test cases
- defining new test cases based on newly established data flow associations
- defining new test cases to satisfy a previously satisfied data flow criterion that is no longer satisfied by existing test cases.

4.1 Choosing a Subset of Existing Test Cases

When selecting regression tests from the previously executed cases, one looks for a small but effective set that will be maximally useful in exposing errors. Data flow can be used to select the specific tests by analyzing the parts of the program that are most "closely related" to the modified code. Using data flow associations as the basis for selecting regression tests directly addresses problems frequently introduced into the code when modifications are made.

Many debugging or updating fixes involve only simple changes to the declaration of a variable, a computation in an assignment statement, or the assignment of a value to an actual parameter. One frequently made error is to change a variable assignment because the assigned expression is incorrect, and not to check that the new expression is appropriate for all the variable's uses. If the result is used in several places, it is easy for the programmer to concentrate on only one or two of those uses, and not to be aware of the uses in other places. The program's cross-reference table can help to avoid this type of problem, but many programmers do not routinely use it. With global variables, there may not even be any facility provided by the language compiler for determining cross-references. The programmer must use an external cross-reference tool (such as Masterscope [Int78] for Lisp or Cscope [Csc84] for C), or put his faith in documentation.

The converse situation occurs when several definition-use associations end in a common node or edge of the flow graph. Such a case means that different declarations or calculations are being used in the same place in the program. The programmer may change the expression in the use of the variable while thinking only of some of the definition-use paths that lead up to the use. The change may fix a problem on one of the paths, but simultaneously introduce a second problem on another. Since in these cases it is important to retest all paths that have been affected by the change, we choose as regression tests those members of the original test set that exercise a definition-use association in which either the definition node or the use node has been modified.

Six types of code change are possible in a modified program:

a) the contents of one or more nodes of the flow graph are changed

b) one or more nodes are deleted from the flow graph

c) one or more nodes are added to the flow graph

d) the predicates on one or more edges of the flow graph are changed

e) one or more edges are deleted from the flow graph

f) one or more edges are added to the flow graph

In case (a) and (d), the syntactic structure of the modified program's flow graph is identical to that of the original flow graph, while the computation within some nodes or on some edges is different. In this case, we can make sure that *the first changed node or at least one changed edge* in any subpath is executed, by using test cases that exercise segments that include the modifications. This strategy, however, does not guarantee executing any subsequent nodes or edges in the subpath, since the first change in the segment may cause the path traversed by the input to change. If the path does in fact change, then a new test case may have to be derived as discussed in Section 4.3.

Note that the path traversed by an input can change either by changing an explicit control statement such as the predicate in an **if-then-else**, or by changing the computation in an assignment statement. The program of Figure 2 illustrates these situations. There are two definition-clear paths with respect to x from the definition of x in node 2 to a c-use of x: (2,3) and (2,4,5). Suppose that for the original program, the test cases $z = 25$ and $z = 64$ were chosen to exercise these paths. If the program is changed by replacing the branch predicate $0 \leq x \leq 5$ with $0 \leq x < 5$, then both of these inputs would execute subpath (2,4,5). In that case node 3 would not be executed, and any changes that might have been made to it would not be tested. On the other hand, if the computation $x := \sqrt{z}$ were replaced by $x := \sqrt[3]{z}$, then both inputs would execute subpath (2,3), and changes to node 4 would not be tested.

In case (d), when a predicate has been changed, the minimal regression test set includes the test cases that execute definition-use associations that terminate with a p-use on

each branch of the predicate that has been changed. These test cases will always test branches that have changed, but there is no guarantee that *all* changed branches will be retested. Consider, for example, the normal two-way branch of an **if** statement. Suppose that the second **if** statement of the program in Figure 2 is changed from

if $(0 \leq x \leq 5)$ **then** ... **else** ...
to
if $(0 \leq x < 5)$ **then** ... **else** ...

This has the effect of changing the predicates on the output branches of node 2 of the flow chart to $(0 \leq x < 5)$ and ($x < 0$ or $x \geq 5$). The result is that the value $x = 5$ is shifted from the true branch to the false branch of the **if**. Hence, if the input $z = 25$ was used in the original program to traverse the p-use association $((2,(2,3)),x)$, in the modified program $z = 25$ will traverse $(2,(2,4),x)$.

In the remaining cases (b), (c), (e), and (f), the modified program has a different flow graph, but it still makes sense to use some of the original test cases. Deleting a node or edge can have various effects on the flow graph, depending on the deleted element's position in the original graph. For case (b), we can select all test cases that exercised definition-use associations in which a deleted node was either the definition or use node. These test cases will bring program control to some predecessor of the deleted node, are are likely to expose errors caused by the deletion. For a simple example, consider deleting node 3 in the program fragment of Figure 5, so the `~p(x)` branch leads directly into node 4. A test case that exercised the original program's c-use association $(3,4,y)$ would test whether y is properly assigned a value in the modified program prior to its use in node 4.

In case (e) an edge is deleted; this edge could have been either labelled or unlabeled. If it was labelled with a predicate, then we select all test cases that exercised p-use associations that terminated on the deleted edge, since in the revised program, those associations now terminate somewhere else. If the deleted edge (m,n) was unlabeled, then we select tests that exercised definition-use associations in which node m was the definition node. These test cases are valuable as regression tests since they take control up to node m, and make use of some computation that occurred in node m.

In cases (c) and (f), new elements are added to the flow graph, creating definition-use associations that did not exist in the original program. In selecting regression tests, we want to execute as many as possible of the original program's definition-use associations that may be affected by the inserted code. To do this, we first determine all immediate predecessor nodes, from the original program, of the inserted nodes and edges, and all labelled edges of the original program that terminate at an inserted node.[3] We can then

[3]Finding the immediate predecessors of a node or edge is a straightforward syntactic operation on the flow graph. If the graph is represented by a set of doubly-linked lists of nodes, the time required is proportional to the number of predecessors.

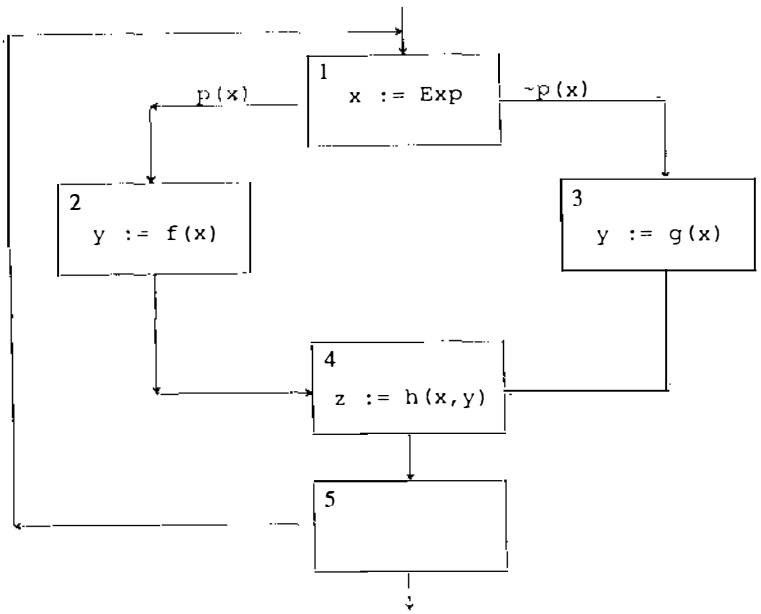


Figure 5: Test Cases for Deleted Nodes

use as a minimal regression test set, those test cases in the Def-Use matrix that exercise the predecessors as definitions, c-uses, or p-uses. A broader set of regression test cases is produced by using the Node and Edge matrices to select all test cases whose traversed paths include any of the predecessor nodes or edges.

4.2 Resatisfying a Previously Satisfied Criterion

It is possible that the modification of a program causes a data flow criterion which was previously satisfied by a given test set to no longer be satisfied. This can occur because new data flow associations have been added by the modifications as considered above. But it is also possible that the *removal* of data flow associations causes a criterion to become unsatisfied. This is because most of the criteria require only that *some* path be exercised from a definition to a given use. When a path is removed or a data-flow association changed, it may then happen that no path of the type required by the criterion is traversed by any of the existing test cases.

Figure 6 provides a simple example of this type of situation. For this program fragment, the definition-c-use associations are $(1,2,x)$, $(1,2,y)$, $(1,3,r)$, $(1,4,r)$, $(2,3,x)$, and $(3,4,w)$. A single test case could satisfy all the c-use associations, by executing the path $(1,2,3,4)$. Suppose this code is modified by replacing the variable x on the left of the assignment statement of node 2 with variable r, so the new statement reads $r := f(x,y)$. After this replacement, $(1,2,3,4)$ is no longer a def-clear path with respect to r, and $(1,4,r)$ is no longer satisfied by the single test case. In this case, therefore,

the original test set turns out to satisfy the new c-use association $(2, 4, r)$, but not the original association $(1, 4, r)$. This situation provides the second basis for the selection of regression test cases. Test data must be chosen to exercise subpaths identified as relevant by ASSET for the selected criterion, but no longer exercised by existing test cases.

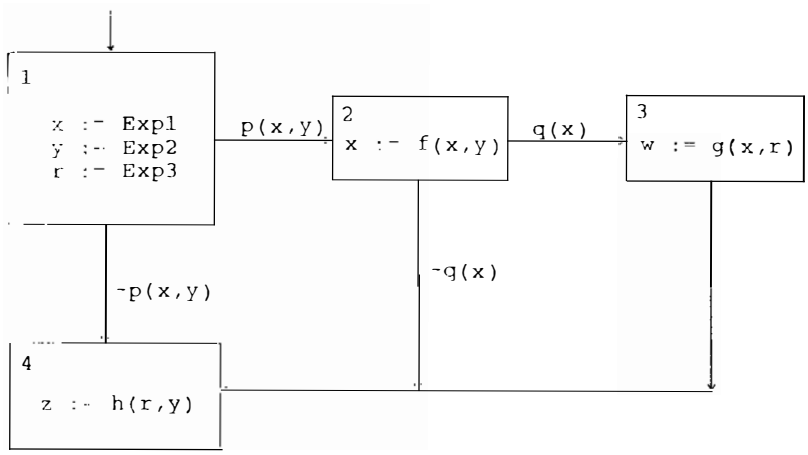


Figure 6: Change in Definition-Use Association

4.3 Selecting New Test Cases

More thorough regression tests can be created for changed software by generating new test cases based on newly established data flow associations. The regression tester selects one of the data flow criteria and uses ASSET to identify data flow associations that have been introduced by the code modifications. Test cases are then created which traverse paths including those associations.

Thus, for new variable definitions, all uses appropriate to the selected criterion are determined by ASSET and test cases must be generated to cover these new definition-use associations. Similarly, for new variable uses, variable definitions are determined and test cases must be generated to cover an appropriate subset of these new definition-use associations as well. Bender [Ben71] proposed a similar strategy for regression testing using what is essentially our all-uses criterion.

This strategy is considerably more costly than reusing existing tests, since the program must be reanalyzed, inputs that satisfy the new associations must be found, and outputs must be manually validated. None of these steps are needed when existing tests are used.

5 Procedure Calls

When paths between procedures are considered for testing purposes, we treat a called procedure Q as a single node in the flow graph of its caller P. A typical regression testing situation occurs when Q has been modified, and one wishes to assure that calls of Q from P still operate properly. We consider the same three bases for regression test data selection as described above for intraprocedural testing.

ASSET analyzes data flow relations across procedure boundaries with respect to both global variables and parameters. Occurrences of a global variable in a procedure are analyzed in the usual manner to determine whether they are definitions or uses. Any variable occurrence in the actual parameters of a call statement is considered to be a use of the variable. In addition, if the formal parameter is an output or reference parameter (a **var** declaration in Pascal), the actual parameter variable occurrence is a definition of the variable.

With these conventions, regression tests are selected with respect to procedure call nodes just as they are for in-line code.

6 Summary and Future Work

Testing is frequently the most poorly designed and inadequately thought-out part of software production. It is treated as a step-child of the development process and considered a boring, uncreative task. If this is the case, then regression testing is the step-child's step-child, and is frequently barely performed at all. In this paper we have described ways of making regression testing a more systematic activity.

Data flow analysis provides a means of deriving test cases that exercise meaningful relations within the code. The regression tests derived from data flow analysis single out and retest the relations that change as a result of code fixes and updates.

A significant task that has not been addressed in this paper is construction of the modified program's flow graph and compilation of the differences between it and the original flow graph. Once the modified flow graph has been produced, use of the ASSET tool greatly simplifies the task of producing regression tests.

A related task is to determine the changed data flow associations after a program has been modified. Rather than reanalyzing the entire modified program from scratch, we will attempt to develop incremental techniques to produce the new associations.

Acknowledgments

Dick Hamlet originally suggested that we consider the application of data flow analysis to regression testing. Marc Balcer, Mike Platoff, and Chris Rumpf raised a number of important issues about data flow-based regression testing, and made many useful suggestions for improving the presentation.

References

[Bar78] J.M. Barth. A practical interprocedural data flow analysis algorithm. *Comm. ACM*, 21(9):724–736, September 1978.

[Ben71] R. Bender. Quality assurance and testing workshop. 1971. Unpublished notes.

[CPRZ85] L.A. Clarke, A. Podgurski, D.J. Richardson, and S.J. Zeil. A comparison of data flow path selection criteria. In *Proceedings of The International Conference on Software Engineering*, pages 244–251, August 1985.

[Csc84] *Cscope Manual.* AT&T, 1984.

[FW85] P.G. Frankl and E.J. Weyuker. A data flow testing tool. In *Proceedings of Softfair II*, IEEE, December 1985.

[FW86] P.G. Frankl and E.J. Weyuker. Data flow testing in the presence of unexecutable paths. In *Proceedings of the Workshop on Software Testing*, pages 4–13, Banff, Alberta, July 1986.

[FW88] P.G. Frankl and E.J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 1988. (to appear).

[FWW85] P. G. Frankl, S. N. Weiss, and E. J. Weyuker. Asset: a system to select and evaluate tests. In *Proceedings of the IEEE Conference on Software Tools*, New York, April 1985.

[Her76] P.M. Herman. A data flow analysis approach to program testing. *The Australian Computer Journal*, 8(3), November 1976.

[Het84] William Hetzel. *The Complete Guide to Software Testing.* QED Info. Sciences, Wellesley, MA, 1984.

[Int78] *Interlisp Reference Manual.* Bolt, Beranek & Newman and Xerox Corp., 1978.

[LK83] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. Software Eng.*, SE-9(3):347–354, May 1983.

[Nta84] S. Ntafos. On required element testing. *IEEE Trans. Software Eng.*, SE-10(6):795–803, November 1984.

[RW82] S. Rapps and E.J. Weyuker. Data flow analysis techniques for program test data selection. In *Proceedings Sixth International Conference on Software Engineering*, pages 272–278, Tokyo, Sept. 1982.

[RW85] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Eng.*, SE-11(4):367–375, April 1985.

[Wey84] E.J. Weyuker. The complexity of data flow criteria for test data selection. *Information Processing Letters*, 19(2):103–109, August 1984.

Testing Shared-Memory Parallel Programs

Andrew H. Sung

Computer Science Department
New Mexico Tech
Socorro, NM 87801

Abstract

This paper addresses the problem of testing shared-memory, synchronous parallel programs. Our model is the parallel random access machine or PRAM. We use a simple, Pascal-based general purpose parallel language for SIMD machine programming. Algorithms designed for both the interconnection machines and PRAM can be implemented in this language.

We propose a scheme for classifying PRAM programming errors. We present several approaches of designing testing coverage criteria for PRAM programs. Various testing coverage metrics, including ones adapted from sequential program criteria and genuine parallel program testing criteria, can be defined using our approaches. The properties of the metrics are discussed. We also present a complete parallel program testing methodology, which incorporates specific coverage metrics and sequential program testing methods. Program testing procedures based on our methods can be easily implemented and integrated into parallel programming environments.

Biographical Sketch of Author

Andrew H. Sung is Assistant Professor and Chairman of the Computer Science Department at New Mexico Tech. His research interests include parallel processing, software engineering, and computational complexity. Sung received his PhD from State University of New York at Stony Brook.

1. Introduction

With the increasing availability of parallel computers and supercomputers, the timely development of methodologies for building and validating parallel software is of crucial importance. Clearly, the tremendous computing power offered by parallel machines can be fully utilized only if we can effectively implement and validate the software for such machines. However, while much has been done on programming systems and environments for parallel computers and supercomputers, little research has focused on validation techniques for parallel programs.

The software validation problem for parallel machines is harder than and different from that for supercomputers. For vectorized/concurrentized programs executed on supercomputers, validation is easier as we can generally assume a correct vectorizing/optimizing compiler and validate the vectorized code by validating the sequential source code or to choose already validated sequential code to begin vectorization with. Except for possible round-off errors (due to the fact that vectorization and concurrentization of sequential code might lead to different round-off error accumulation and therefore give different answers), this simple approach should be sufficient given that the sequential software validation techniques are fairly well established and have been used extensively.

For programs which are truly parallel and are executed on parallel machines containing a large number of processors, the validation problem is more difficult due to the lack of obvious approaches. In this paper we address the issues involved in the validation of parallel programs for shared-memory SIMD machines. Specifically, we investigate the problem of testing programs written for the most powerful model of SIMD parallel computers, the shared-memory, Parallel Random Access Machine or PRAM [Ba88,Qu87]. Our aim is to take an approach general enough to be applicable to a variety of high-level programming methods for SIMD parallel machines. PRAM is chosen because programs developed for interconnection-network type parallel computers can be efficiently simulated on the PRAM and, conversely, PRAM programs can be efficiently mapped to parallel machines based on interconnection networks. The PRAM model therefore provides a good framework for developing and analyzing synchronous parallel programs.

Based on the characteristics of PRAM programs, we first propose a scheme for classifying errors. Two aspects of PRAM programming, processor activation (the sequence of sets of processors activated—PAS) and processor coordination (the sequence of read/write/computation activities of active processor—PCS) are formally described, and PRAM programming errors are characterized in terms of them. Thus, PRAM programming errors are classified as either (1) *processor activation errors*—a program contains a processor activation error if the PAS for some input is different from what is intended, (2) *processor coordination errors*—a program contains a processor coordination error if the PCS for some input is different from what is intended, or (3) *computation errors*—a program contains a computation error if both PAS and PCS are correct but an incorrect result is produced for some input. This error-classification scheme refines the widely used concept of computation and domain errors for sequential software; at the same time, the difficulty of the domain error/computation error dichotomy has been avoided in that our classification guarantees mutually exclusive classes of errors. Our error classification therefore provides a framework for analyzing parallel programming errors.

We then propose a hierarchy of testing coverage criteria for PRAM programs based on input-driven structures like processor activation, processor coordination, read-write sequence, etc. The PAS test coverage criterion, for instance, requires that all feasible PAS (over the entire input domain) be exercised. The relative strengths of these criteria are analyzed. Unlike the common sequential program coverage criteria which are defined based on either the structure or the specification of the program, our criteria combine "white box" and "black box" approaches and concentrate on the salient features of parallel programs, and they constitute basic metrics for measuring testing thoroughness defined specifically for PRAM programs. Coverage criteria based on program flowgraphs and dataflow information are also discussed.

Finally, we present a complete methodology for implementing testing procedures for PRAM programs. This incorporates a parallel to sequential translator, a path finder, and a sequential testing/debugging tool. The basic idea is to choose an appropriate coverage criterion, interactively analyze the test result and choose the next path to be tested, and use the translator and path finder to assist in test data generation. As the translator and path finder translate descriptions of paths within a parallel program to descriptions of paths within an equivalent sequential program, sequential program test generation tools can be utilized to generate tests. Using this methodology, effective and economical parallel program testing/debugging tools can be implemented and integrated into parallel programming environments.

The paper is organized as follows: In section 2 we present a general purpose parallel programming language for PRAM programming (with minor modification, it can also be used for interconnection machine programming). We give in section 3 a classification of PRAM programming errors. In section 4 we define families of testing coverage criteria for parallel programs. In section 5 we present a methodology for testing parallel programs. Section 6 is the conclusion.

2. A Language for SIMD Programming

A general purpose high-level language for SIMD machines should provide means to express parallelism, and to represent data organization and transfer. The rest of this section enumerates those additional features with brief explanations.

2.1 Data Representation and Organization

In a PRAM, all processors (or PEs) are connected to the common or shared memory. One distinguished processor is the control unit or CU. There are three types of variables: PE or local variables, bound locally to each PE; shared or global variables, bound to the shared memory and accessed by all PEs; and CU variables, bound to the CU and accessed only by the CU (physically, CU variables may reside in the shared memory too). Parallel data objects which reside in PEs' local memory or PE objects are identified by the keyword *pe* preceding the data type in their declarations, and variables in the memory of CU (CU objects) are declared as usual. Once declared as PE variables, they are referenced with PE IDs enclosed enclosed in brackets, [ ], after the variable name. The keyword *shared* is used in front of the global data in the shared memory.

2.2 Control of Parallelism

PEs are requested statically using a *network* declaration. The language does not concern itself with the actual number of processors available. A simple *par* <range> *do* <statement> construct is used to represent the activation of different subsets of PEs whose data are referenced in the statement. Those activated PEs execute the statement following *do* in parallel.

Conditional selection of PEs is specified by *where* <predicate> *do* <statement> {*elsewhere* <statement>}, where the predicate involves parallel data in PEs. All the PEs where the predicate evaluates to true execute the statement following *do* while the PEs where it is false execute the statement following *elsewhere*, with the statement in the *do* clause executed before that in *elsewhere*. The *elsewhere* clause is optional.

2.3 Data Transfer through Shared Memory

All data transfers among PEs and between PE and the CU are achieved through writing into and subsequent reading from the shared memory.

This completes the list of our extensions to Pascal. The programming language has been used to code a representative set of parallel algorithms from the literature. Three example programs are given below.

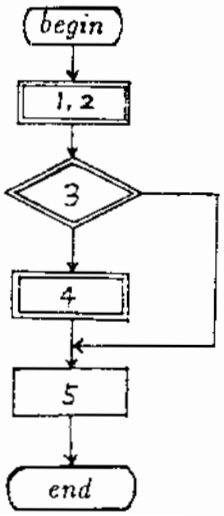
Program 1

procedure OR (n : integer; *var* y : boolean);
{Find the OR of n bits in O(1) time using CRCW PRAM

with common write. See [Ba88] for algorithm. }

```
const
    max = 256;
network
    pe [1..max] : PRAM;
var
    i : integer;
    x : pe boolean;
    M : shared array [1..max] of boolean;

begin
      {Initialization, read input n bits
       into first n cells of shared memory}
{1}   par i := 1 to n do
      begin
{2}   x[i] := M[i];
{3}   where x[i] = 1
{4}   do M[1] := 1
      end;
{5}   y := M[1]
end;
```



Flowgraph of Program 1
see section 4.1 for explanation

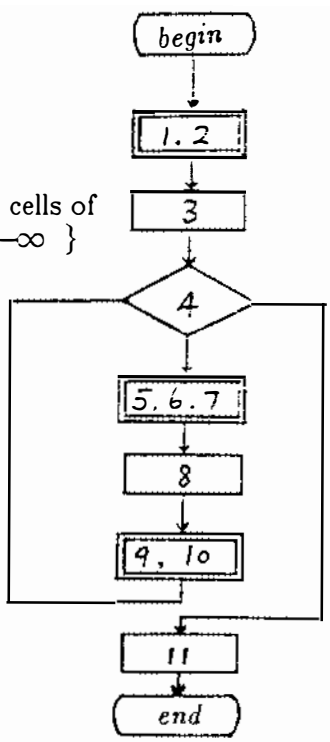
Program 2

```
procedure MAX1 (n,m : integer; var maximum : integer);
{Find the maximum of n integers in O(log n) time using EREW
 PRAM. Assume n = 2^m.  See [Ba88] for algorithm. }

const
    p = 256;
network
    pe [1..p] : PRAM;
var
    i, incr, step : integer;
    temp, big : pe integer;
    M : shared array [1..2*p] of integer;

begin
      {Initialization, read n input numbers into first n cells of
       shared memory, and initialize all other cells to -∞ }
{1}   par i := 1 to n do
{2}   big[i] := M[i];
{3}   incr := 1;
{4}   for step := 1 to m do
      begin
{5}       par i := 1 to n do
          begin
{6}        temp[i] := M[i+incr];
{7}        big[i] := max (big[i], temp[i])
          end;
{8}       incr := 2*incr;
{9}       par i:= 1 to n do
{10}      M[i] := big[i]
      end;
{11}  maximum := M[1]
end;
```

Flowgrpah of Program 2
see section 4.1 for explanation



Program 3

```
procedure MAX2 (n : integer; var maximum : integer);
{Find the maximum of n integers in O(1) time using
 CRCW PRAM with n×(n+1)/2 processors. See
 [Ba88] for a slightly different algorithm. }

const
    p = 256;
network
    pe [1..p,1..p] : PRAM;
var
    i, j : integer;
    x, y, z : pe integer;
    M : shared array [1..p] of integer;

begin
     {Initialization, read n input numbers
      into first n cells of shared memory }
{1}   par i := 1 to n do
{2}   par j := i to n do
     begin
{3}   x[i,j] := M[i];
{4}   y[i,j] := M[j];
{5}   where (x[i,j] <> y[i,j])
{6}   do where (x[i,j] < y[i,j])
{7}        do z[i,i] := 1
{8}        elsewhere z[j,j] := 1;   {every key other than the largest
                                       would have lost a comparison}
{9}   where z[i,i] = 0
{10}  do M[1] := x[i,i]
     end;
{11}  maximum := M[1]
end;
```

Note that there are three models of PRAM—Exclusive Read Exclusive Write (EREW), Concurrent Read Exclusive Write (CREW), and Concurrent Read Concurrent Write (CRCW)—according to whether concurrent reads and writes are allowed; and there are three submodels of CRCW PRAM, depending on how write conflicts are resolved [Qu87,Ba88]. Programs 1 and 2 above work on different models. We will, however, not concern ourselves with the difference between the models. We will also use the term "simultaneous reads (or writes)" instead of "concurrent reads (or writes)" to mean readings (or writings) from (or to) the same location within the shared memory.

3. Errors in PRAM Programs

With the introduction of the new programming features, the widely accepted classification of computation and domain errors [Ho76] for sequential programs is no longer adequate for parallel programs. In the following sections, we identify two classes of errors characteristic of parallel programming for PRAM, namely, incorrect selection and activation of PEs and erroneous reading or writing sequences.

Suppose * denotes any aspect of a parallel program P. We use SPEC(P,*(d)) to denote the correct * of P on input d according to the specification of P. (Equivalently, SPEC(P,*(d)) is the * of the hypothetical, correct version $\hat{P}$ of P on input d.) Further, let IMP(P,*(d)) denote the implemented or actual * of P on d, obtainable through an analysis and/or execution of P. Intuitively, P contains an error whenever SPEC(P,*(d)) differs from IMP(P,*(d)) for any instance of * on some input d. (When P is understood, we use SPEC(*(d)) for SPEC(P,*(d)) and IMP(*(d)) for IMP(P,*(d)).)

3.1 Processor Activation Error

Let P be a parallel program with input domain D. Suppose that P takes t steps to process d $\in$ D, using q logical PEs. (Note that q and t are frequently functions of the input size. Program 2, for example, takes q = n PEs and t = 4m + 3 = $O(\log_2 n)$ steps to find the maximum of n numbers, assuming $n = 2^m$ and ignoring the initialization step.) An input d causes P to activate a fixed sequence of PEs. More specifically, let PAS(d) = $<A_1, A_2, \ldots, A_t>$, a t-tuple, be the "Processor Activation Sequence" of P on input d, where each A_i $(1 \leq i \leq t)$ is a subset of {c,1,2,...,q} and contains the indices of active PEs at step i during P's execution on d (we assume that the CU is indexed with c, and the q PEs are numbered 1,2,...,q). We call each of this subset an "activation snapshot". Further, let PA(d) = {A | A is a component of PAS(d)} be the set of all activation snapshots of P on input d.

One prominent class of errors in SIMD programming is caused by processors being incorrectly activated/deactivated, i.e., a wrong set of PEs being active at some step during the execution of the program. The discrepancy between SPEC(PAS) and IMP(PAS) characterizes this type of errors. That is, a program is said to contain a "processor activation error" whenever SPEC(PAS) $\neq$ IMP(PAS), i.e., SPEC (PAS(d)) $\neq$ IMP (PAS(d)) for some d $\in$ D.

As an example of a processor activation error, consider statement {3} of program 1 erroneously written as "*where* x[i] <> 1", this causes the incorrect subset of PEs to be activated at statement {4}, and thus is a processor activation error.

3.2 Processor Coordination Error

Another notable feature of PRAM programming is processor coordination. Since the processors of a PRAM communicate through a shared memory, readings and writings are coordinated to achieve interprocessor communication; in addition, there are algorithms which specifically exploit the concurrent write features of PRAM. Let P, D, t, q be the same as above. Define PCS(d), the "Processor Coordination Sequence" of P on input d, as $<C_1, C_2 \ldots, C_t>$, where each C_i, called a "PC snapshot", is as follows: (1) If the ith step in the computation is a reading step, i.e., active PEs perform reading from the shared memory, then $C_i = [s_1, s_2, \ldots, s_r]$ such that $s_j \subseteq A_i$ for $1 \leq j \leq r$ and $\bigcup_{j=1}^{r} s_j = A_i$, indicating that r groups of PEs are reading from r disjoint shared memory locations in parallel. PEs in the same group are reading simultaneously from the same location in shared memory. (2) If the ith step in the computation is a writing step, i.e., active PEs perform writing into the shared memory, then $C_i = \{s_1, s_2, \ldots, s_r\}$ such that $s_j \subseteq A_i$ and $\bigcup_{j=1}^{r} s_j = A_i$, indicating that r groups of PEs are writing into r disjoint shared memory locations in parallel. PEs in the same group write simultaneously into the same shared memory location. (3) If the ith step performs an operation other than read or write, then $C_i = |a_1, a_2, \ldots, a_r|$ such that $a_j \in A_i$, i.e., each a_j is the index of a single PE, and $\{a_1, a_2, \ldots, a_r\} = A_i$, i.e., C_i and A_i contain the same subset of PEs. In short, within a PCS(d), [-,-, ..., -] denotes a read step and {-,-, ..., -} denotes a write step, with PEs performing simultaneous reads or writes grouped together, and |-,-, ..., -| denotes a computation step. Also, let PC(d) = {C | C is a component of PCS(d)} be the set of all PC snapshots for input d.

Difference between SPEC(PCS), the correct processor coordination sequence and IMP(PCS), the actual coordination sequence indicates errors. Specifically, we say P contains an "processor coordination error" if SPEC(PAS) = IMP(PAS) but SPEC(PCS) $\neq$ IMP(PCS), i.e., SPEC(PAS(d)) = IMP(PAS(d)) for all d $\in$ D and SPEC(PCS(d)) $\neq$ IMP(PCS(d)) for some d $\in$ D. Thus, processor coordination errors are attributed solely to erroneous PE coordination or erroneous reads or writes, since the PE activation sequence is correct during P's computation on d.

As an example, suppose statement {4} of program 1 were erroneously written as "*do* M[i] := 1", then the active PEs would write to different locations in the shared memory while simultaneous writes are intended; this is a processor coordination error.

3.3 Computation Error

The dichotomy of processor activation error and processor coordination error for PRAM programs can be interpreted as a direct refinement of the concept of domain errors for sequential programs [Ho76]. The other type of errors of sequential programming, i.e., computation errors, can surely occur in parallel programs too. We say P contains a "computation error" if SPEC(PCS(d)) = IMP(PCS(d)) for all inputs d $\in$ D but P(d) $\neq$ $\hat{P}$(d) for some d $\in$ D, where P(d) and $\hat{P}$(d) are, respectively, the outputs of P and $\hat{P}$ on input d. Computation errors are, therefore, caused solely by incorrect assignment statements, incorrect arithmetic operations, etc., when P is given some input d, because both the sequence of PE activations and sequence of processor coordinations are correct during the computation of P on d. (Note that SPEC(PCS) = IMP(PCS) implies SPEC(PAS) = IMP(PAS).)

As an example of a computation error, suppose statement {4} of program 1 were mistakenly written as "*do* M[1] := 0", then error is caused by writing an incorrect value into M[1] and thus a computation error.

One desirable property of our classification is that it makes the three categories of errors mutually exclusive, and thus it forms a formal and useful scheme. Also observe that according to this classification programs that are functionally correct cannot contain computation errors; however, this does not necessarily imply that they are free of processor activation errors or processor coordination errors too. Consider this example of a performance bug: Suppose statement {4} of program 2 were mistakenly written as "*for* step := 1 *to* n *do*", the program will still correctly output the maximum value for every input, and so the error will not be detected by merely checking the output values. Nevertheless, the program contains an obvious error, and in our classification, a processor activation error.

4. Parallel Program Testing Criteria

An essential component of a program testing methodology is a test coverage or adequacy criterion. An ideal criterion should have the following properties: (1) applicable, in the sense that coverage of the criterion can be effectively monitored; further, a test set satisfying the criterion can be effectively constructed. (2) reliable, in that a test set satisfying the criterion has high probability of detecting most errors. (3) cost effective, in that the cost of generating and running a test set satisfying the criterion is acceptable; this implies that the criterion requires only a reasonable number of test cases.

In this section, we present several approaches of defining test coverage criteria for parallel programs and discuss their properties.

4.1 Flowgraph Based Criteria

Three commonly used measures for sequential program testing are statement, branch, and path coverage. To apply these as well as some other structural testing coverage criteria to parallel programs, a flowgraph representation of parallel programs is needed. We can adopt a simple extension of sequential program flowgraphs, as shown in the flowgraphs of the two programs in section 2.

In the flowgraphs, the double-border rectangles are parallel assignment or I/O statements, and the double-border diamonds represent evaluation of the predicates in *where* statements. Single-border rectangles and diamonds are familiar sequential assignment and test of if conditions. The solid lines in the flowgraph represent flow of control of the CU. We say statement i is "executed" by PE j, if statement i involves a variable of PE j. For example, if the input to program 1 is "00101", then statement {4} of program 1 is executed by PE 3 and 5 simultaneously.

To apply the statement, branch, and path testing criteria to parallel programs, we consider a parallel statement (represented by a double bordered rectangle or diamond) "covered" if and only if it has been executed by at least one PE. Thus, the test data of one input "1" can minimally satisfy statement coverage testing of program 1; however, the input "0" does not exercise statement {4} and therefore does not satisfy statement

coverage. The test set consisting of two input data, "0" and "1", satisfy branch coverage of program 1, etc. The advantage of these simple criteria is their consistency with the corresponding sequential testing criteria; that is, like any structural testing criterion for sequential programs, the "structure" of a parallel program can be analyzed from its control flowgraph alone. For example, a path of the program is simply any sequence of alternate nodes and edges of the flowgraph, beginning with the entry node and ending with the exit node.

4.2 Input Based Coverage Criteria

With the presence of new programming language constructs and a variable number of active PEs in parallel programs, the criteria defined in section 4.1 for parallel programs are apparently not adequate. As the execution of a PRAM program entails various PE activation/deactivation and interprocessor communication through shared memory, testing coverage criteria which explicitly require test cases to exercise such events would conceivably be more effective. To define such testing coverage criteria, we first define a path of a parallel program in such a manner that the description of a path contains information about PE activation.

4.2.1 Path Coverage Testing

Let P be a parallel program with s executable elementary statements, numbered 1 through s. (An elementary statement is one that can be executed in one step. E.g., in our programming language, a *where* <B> *do* <S1> *elsewhere* <S2> is the composition of three elementary statements if both <S1> and <S2> are simple assignment statements.) We define Path(d), the path of P exercised by input d, to be $(l_1,A_1)(l_2,A_2)...(l_t,A_t)$, where $1 \leq l_i \leq s$ $(1 \leq i \leq t)$, and l_i represents the ID of the statement being executed at the ith step; A_i is the ith coordinate of PAS(d), denoting the set of active PEs at the ith snapshot during the computation of P on d. This definition of path of parallel programs is compatible with that of sequential programs; however, as it represents an actual path traversed on a specific input, all infeasible paths are excluded.

Once paths are defined for parallel programs, the corresponding path testing criterion can be formulated as follows: Select test set $T \subseteq D$ such that $\{Path(d) \mid d \in D\} \subseteq \{Path(d) \mid d \in T\}$. Path testing requires the generation of a test set containing enough test cases such that every path that can possibly be traversed by an input during P's computation be exercised at least once. This criterion is not practical as programs containing loops can have an infinity of paths.

4.2.2 Criteria Based on PAS and PCS

Based on the concept of processor activation and processor coordination, a family of testing coverage criteria can be defined. These criteria, along with the path coverage criterion of the previous section, are based on input-driven structures of the program and cannot be derived solely from the program or its specification.

(1) PCS (Processor Coordination Sequence) Testing:
Select test set $T \subseteq D$ such that $\{PCS(d) \mid d \in D\} \subseteq \{PCS(d) \mid d \in T\}$.
PCS testing requires that every possible processor coordination sequence during P's execution over all input points be exercised at least once.

(2) PAS (Processor Activation Sequence) Testing:
Select $T \subseteq D$ such that $\{PAS(d) \mid d \in D\} \subseteq \{PAS(d) \mid d \in T\}$.
PAS testing requires that every possible processor activation sequence during the computation of P over its entire input domain be driven at least once.

(3) PC (Processor Coordination) Testing:
Select test set $T \subseteq D$ such that $\{PC(d) \mid d \in D\} \subseteq \{PC(d) \mid d \in T\}$.
PC coverage testing requires that every possible processor coordination during the execution of P over all input points be invoked at least once.

(4) PA (Processor Activation) Testing: Select $T \subseteq D$ such that $\{PA(d) \mid d \in D\} \subseteq \{PA(d) \mid d \in T\}$.

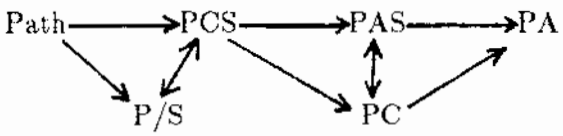
PA testing requires that every possible processor activation snapshot during the computation of P over its entire input domain be tested at least once.

(5) P/S (Processor/Statement) Testing: Select $T \subseteq D$ such that $\{E(d) \mid d \in D\} \subseteq \{E(d) \mid d \in T\}$, where E(d) the executability matrix, is a q × s matrix such that

$$E(d)[i,j] = \begin{cases} 1, & \text{if } X(d)[i,j] \geq 1 \\ 0, & \text{if } X(d)[i,j] = 0 \end{cases}$$

where X(d), the profile matrix, is a q × s matrix such that X(d)[i,j] = number of executions of statement j by processor i, when P is run on input d. P/S testing requires a test set T containing enough test data such that if it is possible for processor i to execute statement j when P is given some input $d \in D$, then there is a $d \in T$ which drives processor i to execute statement j.

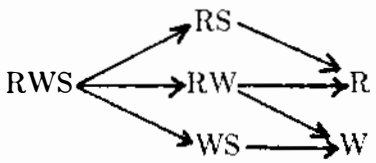
For two testing coverage criteria C1 and C2, C1 (strictly) subsumes C2 or C1 $\rightarrow$ C2, if every test set T that satisfies C1 also satisfies C2 but not vice versa. C1 and C2 are incomparable or C1 $\leftrightarrow$ C2, if neither one subsumes the other [RaWe82]. The following lattice diagram shows the relative strengths of the path testing criterion and criteria (1) - (5) above.



4.2.3 Criteria Based on Read and Write

Reading from and writing into the shared memory being a conspicuous feature of PRAM programs, an alternative approach to define test coverage criteria based on input-driven structures is to concentrate on the read/write events of PRAM programs. Let RWS(d) = $<X_1, X_2 \ldots, X_t>$, where each X_i, called a "Read/Write snapshot", is as follows: If the ith step in the computation is a read or write step, then $X_i = C_i$ = the ith coordinate of PCS(d); otherwise, $X_i = \varnothing$. That is, RWS(d) concentrates on the read/write portions of PCS(d). Let RW(d) = {X | X is a component of RWS(d)} be the set of all R/W snapshots. Similarly, RS(d) and WS(d) consist of, respectively, the read and the write portions of PCS(d); R(d) and W(d) are respectively the set of all reads and the set of all writes.

Corresponding to the definitions above, a hierarchy of six test coverage criteria can be formulated. Their relative strengths are indicated in the diagram below. Compared with the criteria in section 4.2.2, the relationship are PCS > RWS and PC > RW, most other pairs of metrics are incomparable.



4.3 Data Flow Based Testing

Useful data flow testing metrics are necessary components of complete testing methods for parallel programs, especially because the correctness of parallel programs depends on correct data flow interactions even more critically than sequential programs do. In this section, we present three simplifying measures to apply the data flow based, sequential program testing metrics to parallel programs: (1) Treat a PE object in different processors as one entity. (2) Monitor the data flow activities of CU objects only. (3) Combine data dependence and control dependence. The merits and disadvantages of this

approach are briefly discussed.

4.3.1 Criteria Based on Unifying PE Variables

In this approach, we consider each PE variable in a parallel program as a single variable rather than a collection of variables bound to different PEs; thus, PE variables can be treated in the same way as arrays of sequential programs. In program 2, for example, definition of PE variable "big" at statement {7} is considered live at {10}, definition of variable "temp" at {6} is live at {7}, etc. With this unification of PE variables, liveness of each (CU and PE) variable definition at each statement can be determined statically. Therefore, algorithms that were designed for computing this for sequential programs (see [MuJo81]) can be used to obtain the same information for parallel programs; and anomaly detection tools based on data flow information for sequential programs can be utilized for parallel programs with minimal modification.

With the concept of liveness of definitions extended to parallel programs, concepts such as "data environment", "data context", and "ordered elementary data context" for sequential programs [LaKo83] can be similarly extended to parallel programs. With theses extended definitions, data flow based static testing coverage metrics for sequential programs can be applied directly to parallel programs.

4.3.2 Criteria Based on CU Variables

In this approach, we concentrate on the data flow activities of CU variables and ignore that of PE variables. In most parallel programs, CU variables are used to control the execution of loops which contain parallel statements (i.e., *par* and *where* statements for our language of section 2) or to store results of computation. Exercising a particular program path to test a particular data flow activity of CU variables inevitably exercises and tests other constructs.

For example, a *for* loop index (CU) variable is defined (initialized) before the loop is first entered, it is used (incremented) at the end of the loop, and used (in a predicate) to decide if traversal is to be continued at the beginning of the loop. Therefore, a metric such as "requiring that the definition-use chain of length 2 of every *for* loop index variable be tested at least once" necessarily forces the body of the *for* loop to be tested at least once.

4.3.3 Combined Data Flow and Control Flow Criteria

Static program analysis methods based on combining data flow and control flow information have been proposed recently in reference [Ko87]. The approach is to use a program dependence graph to model the data and control dependences between program instructions. Taking the simplifying measures of unifying PE variables or concentrating on CU variables, we can extend the concept of program dependence graph and apply it to parallel programs. Thus, algorithms and tools based on program dependence graphs and developed for sequential programs [KoLa85] can be used for parallel programs too. Static testing metrics based on combination of data dependence and control dependence, e.g., requiring each control dependence and each data dependence to be tested at least once, can also be defined.

4.3.4 Discussion

There are obvious advantages of using static data flow metrics for measuring testing thoroughness for parallel programs: these metrics are essentially the same as that for sequential programs, and therefore data flow analysis tools for sequential programs can be readily used. The major disadvantage, however, is that these strategies are rather weak. For examples, it can be proved that the metric of testing all live definitions from the data environment of every instruction or block of [LaKo83] is incomparable with the branch coverage criterion; the example CU variable based metric in section 4.3.2 is strictly subsumed even by branch coverage. For many parallel software applications requiring high level of reliability such as in real time systems, such structural metrics are not adequate.

5. Practical Testing Strategies

In this section, we present a parallel program testing methodology which incorporates the coverage metrics defined in section 4. Two methods, incorporating, respectively, the flowgraph based metrics and the input based metrics, are described as examples. A preprocessor consisting of a translator which translates parallel programs into sequential programs and a path finder which, for a path of a parallel program, finds the corresponding path in the sequential program is a major component of our methods. The testing procedures also incorporate a sequential program test data generation mechanism which the programmer/tester is already familiar with. Thus, the methods can easily be implemented.

5.1 Flowgraph Based Structure Coverage Testing

To perform a statement (or branch) testing of a parallel program P, a set of paths S $= \{p_1, p_2, ..., p_k\}$ is first selected to cover all statements (or branches) of P. P and S are then input to the translator and path finder. The outputs are an equivalent sequential program $\hat{P}$, and the corresponding paths $\hat{S} = \{\hat{p}_1, \hat{p}_2, ..., \hat{p}_k\}$. Once $\hat{P}$ and $\hat{S}$ are obtained, a preferred sequential program test generation method can be used to derive the desired test set $T = \{d_1, d_2, ..., d_k\}$. Then, the test data can be executed with P. With proper instrumentation (for example, inserting counters at each decision to decision or d-d path to monitor branch coverage [Hu75]), the above procedure can be carried out incrementally. That is, first, translate P into $\hat{P}$, and then iteratively use the path finder and a chosen sequential program test generation method to obtain successive test cases, until all statements (or branches) have been exercised.

To carry out path testing, it is necessary to first apply an equivalence relation to partition the infinite set of paths (or the large number of paths) into finite equivalence classes. (For example, Howden's boundary-interior method [Ho75] can be directly extended to parallel programs.) Then the preprocessor can be used to guide test data generation until a representative path from each class has been covered by a test case.

The method described above can also be used to conduct testing with a static data flow coverage criterion like the ones described in section 4.3.

5.2 Input Based Structure Coverage Testing

The input based structure coverage criteria of section 4.2 are not directly applicable as they may require an infinity of test cases. To perform a testing to cover a particular input based structure Q (e.g., Q = PCS, so that Q coverage testing is PCS testing), it is therefore necessary to find an appropriate equivalence relation among instances of Q to reduce the corresponding infinite criterion to a finitary, satisfiable one. (Formal methods of defining equivalence deserves further investigation, even though it seems unlikely that a generic and meaningful equivalence relation among parallel program paths can be found, due to the large variety of data structures, control structures, and interprocessor communication patterns that can be present. To reduce the domain of Q into finite classes, a useful equivalence relation will have to be defined from the specification and the program by the tester.)

After an appropriate equivalence relation is found and applied to an infinite input based criterion Q, it becomes finitary and thus satisfiable, because a test data which covers an instance of Q covers all equivalent instances. Let's use Q_c to denote this practical Q coverage criterion. The testing of a parallel program P can then proceed according to the following procedure:

(1) P is instrumented to monitor the coverage of Q_c. An equivalent sequential program $\hat{P}$ is then produced by the translator.

(2) Choose an input d from P's input domain, say randomly.

(3) Execute d with P (or $\hat{P}$). The result is analyzed to determine the path and the instance of Q which were exercised by d.

(4) Determine the next path p of P to be traversed. This path, of course, is to exercise an instance of Q which has not been covered by previous test data.

(5) Input p and $\hat{P}$ to the path finder to produce the corresponding sequential path $\hat{p}$ of $\hat{P}$.

(6) A pre-selected sequential software test data generator is used with $\hat{P}$ and $\hat{p}$ to generate the next test case d.

(7) Repeat steps (3) - (6) until the criterion Q_c is satisfied.

The method described above for input based coverage testing incorporates the essential aspect of Kundu's method of generating tests for sequential [Ku79], namely, reversing the roles of path selection and test data generation. For examples of the above methods applied to parallel programs, refer to [FaSu88].

5.3 Input Space Partitioning

As explained in the previous section, the tester is responsible for finding a useful equivalence relation to partition the infinite instances of an input-driven structure into finite classes. Though it seems unlikely that a single relation can be found and applied universally, often heuristics can be used to serve the purpose. Another testing method is to define an equivalence relation to partition the input space itself into a desirable number of classes, and then choose one representative test data from each class.

In program 2, say, we can define equivalence in terms of the degree of parallelism exhibited during execution of the program. As an example, the input "n=1, m=0" causes the *for* loop beginning at statement {4} to be skipped altogether and thus no parallelism is exhibited. The input "n=2, m=1" causes the *for* loop to be traversed exactly once and hence the *par* statement at {5} executed with two PEs exactly once, thus a minimal parallelism is exhibited. The input "n=4, m=2" causes the *for* loop to be traversed twice and hence the *par* statement at {5} will be executed twice by four PEs, thus a "nonminimal" parallelism is exhibited. If a small number of test cases is desired, then all inputs with $n \geq 4$ can be classified as equivalent in terms of exercising nonminimal parallelism, and thus the three described test data would suffice. This idea is similar to the boundary-interior partition method for sequential programs [Ho75]. Usually more thorough testing is needed, and the partition method can be refined to any desired level of fineness.

Consider program 1 for another example. The algorithm has O(1) time complexity and therefore the method of partitioning based on exhibited parallelism is not as desirable. Then we may consider the input of n bits as representing an integer, and define two inputs to be equivalent if they are congruent modulo a certain number, e.g., consider inputs x and y to be equivalent iff $x \equiv y \bmod 20$ if 20 test cases are planned. Again, this can be refined to any desired level.

5.4 Use of Sequential Programs

Much research effort has gone into building translators or compilers which take sequential programs and detects and generates parallel codes for the target parallel machine [AlKe84, Ku82]. Conversely, a parallel-to-sequential translator can be used to simulate the execution of parallel programs on a sequential computer if desired. Such a translator has been implemented for the execution of the programs written in our extended Pascal.

For synchronous parallel programs, the same input should always produce the same output, therefore they can be translated, one-to-one, into sequential programs. As the sequential program gives exactly the same result for every input point as the corresponding parallel version, one is functionally correct if and only if the other is. Thus the parallel program can be validated by testing either the original or the translated sequential program. Obviously, this methodology is not limited to a specific sequential testing strategy, and any of the existing testing metrics and procedures for sequential programs can directly be used. For example, a "parallel program mutation testing system" may be

implemented by simply combining a translator and a sequential program mutation system [Bu81]. This is the most straightforward and readily adoptable methodology, and can be used as a transitional means while new techniques for parallel software validation evolve. Also, with this method, less expensive sequential computers can be used for software validation purposes.

This method, however, is far from being adequate, and should be considered only as a minimal tool. Moreover, it has several disadvantages. First, the method evidently does not consider the structural aspects unique to parallel programs, hence could ignore some of the likely sources of errors. Secondly, as explained at the end of section 3, programs which contain errors might still perform the correct input-output transformation. As it is hard to obtain timing information about the parallel program by simulation, this method would fail to detect, at least for the programming language presented here, most of the errors that are not evidenced by incorrect output values.

5.5 The Preprocessor: Translator and Path Finder

We have implemented the preprocessor for the execution and testing of parallel programs written in the Pascal-based parallel language. The preprocessor consists of two components, namely the translator and the path finder. The translator takes parallel programs as its input, and produces sequential Pascal programs as well as data files for the path finder. The path finder takes a set of path specifications for a parallel program as its input and produces the corresponding paths in the translated sequential program. Implementation details of the preprocessor is discussed in [FaSu88].

6. Conclusion

We have presented a scheme for classifying PRAM programming errors and proposed various approaches of deriving testing coverage metrics for shared-memory parallel programs.

The approaches based on extending the sequential methods can be used to define testing metrics which are easily understood and applicable. On the other hand, they tend to define weak metrics due to the fact that they overlook the features unique to parallel programming.

The criteria based on input-driven structures, combined with the proposed translator/path finder approach to testing provide a viable methodology. The advantages of this methodology are [1] practical—the preprocessor, consisting of the translator and path finder, is easy to implement and takes only linear time, [2] economical—sequential program test generation tools, which programmers/testers are already familiar with, can be "plugged in", thus our software investment in testing/debugging tools for sequential programs can be utilized in a parallel environment and personnel training effort can be minimized, and [3] immediately usable—it can be readily used while various programming methodologies evolve and new techniques for parallel program validation are developed.

Testing has been the most widely used method for validating sequential software and there is growing attention to the theory and techniques of testing [Ha88]. In view of the increasing applications of parallel computing and the lack of effective methodologies for validating parallel software, the presented approaches and methods provide useful directions for further research and development.

Acknowledgment

I wish to thank the referees for helpful comments and suggestions. I also thank Min Fang for implementing the preprocessor described in section 5.5. Support for this work from the Research and Development Division of New Mexico Tech is gratefully acknowledged.

References

[AlKe84] Allen, J.R. and K. Kennedy, *A Parallel Programming Environment*, Computer Science Dept. Technical Report TR84-8, Rice Univ., July 1984.

[Ba88] Baase, S, *Computer Algorithms, 2nd Edition*, Addison-Wesley, 1988.

[Bu81] Budd, T.A., *Mutation Analysis: Ideas, Examples, Problems and Prospects*, in B. Chandrasekaran and S. Raddichi, eds., Computer Program Testing, North-Holland (1981) 129-148.

[FaSu88] Fang, M. and A.H. Sung, *A Preprocessor for Testing Parallel Programs*, Computer Science Technical Report, New Mexico Tech, 1988.

[Ha88] Hamlet, R. et al., *Special Section on Software Testing*, Comm. of the ACM, vol. 31, no. 6 (1988) 662-695.

[Ho75] Howden, W.E., *Methodology for the Generation of Program Test Data*, IEEE Trans. on Computers, vol. C-24 (1975) 554-560.

[Ho76] Howden, W.E., *Reliability of the Path Analysis Testing Strategy*, IEEE Trans. on Software Engineering, vol. SE-2, no. 3 (1976) 208-215.

[Hu75] Huang, J.C., *An Approach to Program Testing*, ACM Computing Surveys, vol. 7, no. 3 (1975) 113-128.

[Ko87] Korel, B, *The Program Dependence Graph in Static Program testing*, Information Processing Letters, vol. 24 (1987) 103-108.

[KoLa85] Korel, B. and J. Laski, *A Tool for Data Flow Oriented Program Testing*, in SoftFair II - 2nd Conf. on Software Development Tools, Techniques, and Alternatives (1985) 34-38.

[Ku79] Kundu, S., *SETAR - A New Approach to Test Case Generation*, INFOTECH State of the Art Report, Software Testing, Infotech Intl. Ltd (1979) 163-186.

[La82] Laski, J.W., *On Data Flow Guided Program Testing*, ACM SIGPLAN Notices, vol. 17, no. 9 (1982) 62-71.

[LaKo83] Laski, J. and B. Korel, *A Data Flow Oriented Program Testing Strategy*, IEEE Trans. on Software Engineering, vol. SE-9, no. 3 (1983) 347-354.

[MuJo81] Muchnick, S.S. and N.D. Jones, *Program Flow Analysis : Theory and Applications*, Prentice-Hall, 1981

[Qu87] Quinn, M.J., *Designing Efficient Algorithms for Parallel computers*, McGraw-Hill, 1987.

[RaWe82] Rapps, S. and E.J. Weyuker, *Data Flow Analysis Techniques for Program Test Data Selection*, Proc. of 6th Int'l Conf. on Software Engineering (1982) 272-278.

Debugging

"On the Cost of Back-To-Back Testing"
M. A. Vouk, North Carolina State University

"An Execution Backtracking Approach to Program Debugging"
Hiralal Agrawal and Eugene H. Spafford, Software Engineering Research Center, Purdue University/University of Florida

"A Debugging Assistant for Distributed Systems"
Daniel Hernandez, Technische Universitaet Muenchen;
Laveen Kanal and James Purtilo, University of Maryland at College Park

On the Cost of Back-To-Back Testing*

Mladen A. Vouk

Department of Computer Science, Box 8206
North Carolina State University
Raleigh, NC 27695-8206

Abstract

Developing two or more functionally equivalent versions of software and comparing their outputs against each other offers a potentially very simple way of testing software with a large number of test cases. One of the names for the technique is back-to-back testing. This paper focuses on the economics of back-to-back testing. We use two models to explore the cost-effectiveness of back-to-back testing. The key parameters are the degree and the nature of the inter-version failure correlation, the per failure identification effort required during single version program testing, the intensity of back-to-back testing false alarms, and the shape of single and multiversion reliability growth functions. The conditions under which back-to-back testing can be cost-effective are discussed. The available experimental information on back-to-back testing is briefly reviewed.

Biographical Note

Mladen A. Vouk received B.Sc. and Ph.D. degrees from University of London (U.K.) in 1972 and 1976 respectively. From 1978 until 1984 he was with the University Computing Centre, University of Zagreb (Yugoslavia), where he worked in the field of numerical software, programming languages, and biomedical real-time data acquisition and processing. In 1985 he joined North Carolina State University as an Assistant Professor of Computer Science. His areas of interest include software reliability and fault-tolerance, software testing, numerical software, and mixed language programming.

Dr. Vouk is a member of ACM, IEEE Computer Society, Sigma Xi, and American National Standards Institute Technical Committee X3T2 on Data Interchange. He is the current secretary of the Working Group 2.5 on Numerical Software of the International Federation for Information Processing.

* This research was supported in part by NASA grant NAG-1-667.

On the Cost of Back-To-Back Testing*

Mladen A. Vouk

Department of Computer Science, Box 8206
North Carolina State University
Raleigh, NC 27695-8206

1. Introduction

High reliability is of paramount concern in many applications. Often these applications are also functionally very complex, and unless proof of program correctness is feasible, or correctness can be demonstrated in some other way, validation based on execution of many test cases may consume a large part of the development effort. In this context the development and testing of two or more functionally equivalent versions of a program against each other has an enormous appeal because it offers a potentially very simple way of checking for the correctness of a very large number of test cases. The technique appears under different names such as dual code or program testing, parallel programming and testing, distinct software testing, comparative or comparison testing, diverse system testing, back-to-back testing, and multiversion testing [e.g. Fis75, Gil77, Ram81, Pan81, Ehr85, Bis86, BrK86, Shi88, Vou88]. We will use the term back-to-back testing. The effectiveness of the technique has been discussed by a number of authors. The cost has been examined in more detail by by a few [Gil77, Pan81, Ehr85, Sag86, Bis86], although a general opinion about the cost-effectiveness of the approach can be found in almost any paper discussing back-to-back testing.

The aim of this paper is to generalize and extend the cost of diversity model proposed by Ehrenberger [Ehr85, Sag86], and introduce a new model based on software execution time reliability growth. Both models incorporate the influence of inter-version failure correlation.

In section 2 of this paper we present a brief overview of back-to-back testing issues. In section 3 we discuss a simple cost model of multiversion testing process. We use the model to highlight the basic properties of back-to-back testing. In section 4 we present a more complex multiversion testing cost model based on an execution time reliability growth model. In section 5 we summarize the results and offer an evaluation of back-to-back testing.

* This research was supported in part by NASA grant NAG-1-667.

2. About back-to-back testing

Back-to-back testing involves pairwise comparison of the responses from several functionally equivalent software versions as a means of judging the correctness of the outputs. Whenever a difference is observed among responses, the problem is thoroughly investigated in all versions and for all test cases where even one component answer differs. Debugging of the programs is an integral part of the process. All answers have to be identical to within a tolerance if a "no failure" event is said to occur. We will assume that back-to-back testing is used to help develop and test one final version of the software.

We will call a set of k functionally equivalent program versions a k-tuple. For tractability we will assume that at any time during back-to-back testing all k independently developed versions have a similar failure probability equal to p. Theoretical [Eck85], and experimental [e.g. Sco84, Kni86] work indicates that independently developed version may not fail independently, although, again in theory [Lit87], it should be possible to achieve independence through use of sufficiently diverse development methodologies. At worst, all versions contain the same similar faults (fault span is k) which makes them fail coincidentally for all inputs, i.e. the probability of coincident failures is now p. Back-to-back testing fails to detect a coincident failure of all the versions (failure span is k), if all the answers are identical to within a numerical tolerance (e.g. there is an identical fault in all versions because the programmers made the same mistake), or if the output space is binary. Usually the conditional probability of obtaining an identical and wrong answer, given that a coincident failure of all the components has occurred, is less than one. At best, there are no coincident failures involving all the versions at all, or if they occur not all the answers are identical. This can happen if the inter-version failure correlation is negative to the point of partial or complete mutual exclusion of version failures.

Back-to-back testing has been receiving mixed reviews. On the one hand, the technique requires relatively high initial expense of developing two or more functionally equivalent programs, an adequate (and potentially expensive) test bed [e.g. Ram81, Ehr85], and, worst of all, its efficiency may be seriously impaired if the inter-version failure correlation is large [Sag86, Bis86, Vou88]. Unfortunately, there is strong evidence that correlated failures resulting in identical and wrong answers from two or more functionally equivalent components can occur under currently employed development strategies [e.g. Sco84, Vou85, Bis86, Kel88, Shi88]. On the positive side, back-to-back testing permits extremely extended testing by program execution with minimal human supervision which offers a promise of very high reliability at an affordable cost [Gil77, Ehr85, Sag86, Vou88]. There are also indications that back-to-back testing is effective in discovering faults

not detected by other testing techniques [e.g. Pan81, Bis86, Shi88].

In [Vou88] it is shown that the probability, $P_D(k)$ (per test case), that back-to-back testing of a k-tuple detects a failure can be expressed as

$$P_D(k) = 1 - [\gamma P(1+) + P(0)] \qquad (2.1)$$

where $\gamma P(1+)$ represents the probability of "an identical and wrong answer from k versions", $P(1+)$ is the probability that "one or more of the k-versions fail coincidentally", γ is the conditional probability for the event "k versions return an identical answer given that one or more of the k versions have failed", and $P(0)$ represents the probability that all k versions succeed (i.e. zero fail). The correlation, if any, among failures encompasses both the correlation due to similar or common-cause faults and that due to the cardinality of the output space. If the correlation due to similar faults is zero but the output space is binary (the answer is either correct or incorrect without distinction among possible incorrect answers), then the expression reduces to $P_D(k)= 1- [p^k+(1-p)^k]$. If there is only a single similar fault in all the versions then $P(1+)=p$, $P(0)=(1-p)$, and γ must be less than one for the fault to be detectable by back-to-back testing.

Ideally, back-to-back testing has the power to detect all the failures the first time they occur ($\gamma = 0$), or at least after the same failures have repeated several times ($\gamma < 1$, $P(1+)>0$). It can be shown that the number of random test cases needed to guarantee detection of a failure at α confidence level is

$$T = \frac{\ln(1-\alpha)}{\ln(1-P_D(k))} \qquad (2.2)$$

Given a particular similar fault, the larger the failure correlation, the more test cases have to be executed before that fault is detected. In the extreme, when correlation is 100% ($\gamma=1$, $P_D(k)=0$), the fault cannot be detected by back-to-back testing. It can also be shown that if there are no correlated faults which have failure span over all versions with $\gamma=1$, then using more than about 4 versions with $p \leq 0.05$ results in rapidly diminishing returns in terms of the failure detection efficiency [Vou88]. As the reliability of the versions grows (hence, p reduces with testing), the effectiveness of having many versions running reduces. In an ideal situation, where there is no failure correlation, only two versions would be needed to detect all the faults. However, in practice it is necessary to consider the trade-off between the number of versions, the potential span of correlated faults (if any), and the number of test cases we can actually run within the development schedule. For example, using only two versions is reasonable only if the risk of missing a correlated fault of span two is acceptable,

otherwise more than two versions need to be developed. Since there is no guarantee that development of additional versions will reduce failure correlation, the best policy is to develop the software for maximal algorithmic diversity, use back-to-back testing to detect as many uncorrelated faults as possible, and follow this by post-multiversion testing of a single version, aimed at the faults missed by back-to-back testing. Empirical evidence suggests that at least 80% of the faults are uncorrelated [e.g. Vog88, Bis86, Shi88, Bis88]. Hence, if back-to-back testing offers a considerable saving with respect to single version testing for over 80% of the faults, then it may still be cost-effective to test for the residual faults using other, more labor intensive, single version methods.

3. A Simple Model

The general form of the cost model we will use is

$$E_{sys} = a_{req1}E + k\, a_{req2}E + k\, a_{dev}E + E_{test}. \qquad (3.1)$$

E_{sys} is the total effort required to develop the system (whether it consists of a single version or multiple versions), $a_{req}=a_{req1}+a_{req2}$ is the fraction of some "basic" development effort expended in the requirements phase of the life cycle, and k is the number of versions being built. E is the "basic" effort for a single stand-alone version (includes design, coding, unit testing, integration and some system testing). The "basic" effort , for example, could be estimated using the COCOMO model [Boe81]. Note that E does not include the effort associated with the requirements, i.e. $E = a_{dev}E + E_{test}$, where $(1-a_{dev})$ is the fraction of the effort devoted to "basic" system testing. The multiversion development and testing process begins with writing of multiple manufacturer specifications ($k\, a_{req2}E$) on the basis of a single customer requirements document. This is followed by multiple design, coding, unit testing and integration effort ($k\, a_{dev}E$), and finally by the (system) testing phase. The "basic" effort incorporates all the actions that are necessary for a particular type of software at the given level of quality, including planning, construction, execution and evaluation of as many test cases as can be fitted into the "basic" calendar time schedule and testing effort, $(1-a_{dev})E$.

The principal differences arise at system testing time. Execution of test cases without careful evaluation of the outputs is an exercise with very minimal returns. Unless the failures are self-reporting there is no way of determining whether the outputs were correct or incorrect. Investigations of the reliability growth process have shown that the major part of the testing effort may be the identification of failures [e.g. Mus87]. In the simplest case the failures may be self-reporting through serious and easily noticeable interruption of the computer service. For example, if we are testing an

operating system and are only interested in the fatal run-time failures that result in system re-boots, then it is easy to detect the failures. On the other hand, if the computations performed by the software are very complex the correct result may not be easy to compute, and manual verification of the acceptability of each output may require considerable effort. Sometimes it is possible to construct (usually very complex) consistency checks based on the known values of the input data and mathematical characteristics of the employed solution algorithms. This acceptance test may be quite efficient in automatically detecting failures [e.g. Pro88], but it may also require construction of a complex simulator in order to provide an adequate approximation of real time events and input data. If the effort required to verify the correctness of an output is minimal (almost self-reporting failures), then it is possible that a sufficient number of test cases can be run and evaluated within the framework of the "basic" testing effort to achieve the target reliability.

To see this we first make a very simplistic assumption about the reliability growth process (we use a more sophisticated approach in section 4). As the testing progresses and more test cases are executed our estimate of the reliability (R) of a system increases. It can be shown [e.g. Dur84, Ehr85] that if T representative random test cases are executed and no failures are found, then an upper bound, p_u, on the failure probability, p = 1-R, of the system, at α confidence level, is given by the following expression:

$$p_u = 1 - (1-\alpha)^{1/T}. \qquad (3.2)$$

We will assume that the the number of test cases we need to run to achieve a target reliability is at least T, where T is computed from the above equation. Since in practice, neither failure detection effectiveness, nor failure repair activities are always perfect it is not be surprising that frequently more than this number of test cases is needed to reach the desired target reliability (experiments indicate that efficiency of failure detection using classical, single program testing, strategies can be as low as 80% [Bis86]).

What is then the number of test cases that can be developed and run for the "price" of the "basic" testing effort? Depending on the complexity of the function/problem that is being solved and the nature of the failures we are interested in we may be able to prepare, execute and evaluate only a limited number of test cases during the software development phase (e.g. for non self-reporting failures: 10 test cases per person-hour, or 5 tests per person-day, or maybe one test in four days). Of course, it is not just the quantity of test cases that is important, but also the quality. Therefore, for the "basic" testing effort, it may be possible to produce an effect (reliability) equivalent to $T_{dev} = (1-a_{dev})E/X_{case}$ representative random test cases, where X_{case} is the effort per equivalent test case (e.g. 100 actual cases may result in software reliability R=0.997 which is about 1000

"equivalent" cases using (3.2) with α=0.95). If higher reliability is required, the testing effort will grow in excess of this "basic" effort, with part of the effort going to activities that may not be directly associated with the testing [e.g. Boe81].

With back-to-back testing, once the testing environment has been set up at the cost of a_{bed} units of the "basic" cost, the execution of test cases and identification of failures is almost free. A large number of test cases can be generated (e.g randomly) and run, limited only by the available computer time, computer speed and calendar schedule. Of course, in the case of an k-tuple the cost of actually identifying the faults, after a failure has been diagnosed, and repairing them will be about k times that for a single version since failure correction activities have to be applied to all versions of a k-tuple. In the current model the cost of failure identification and failure correction (the latter consisting of fault identification and fault correction activities [Mus87]) are fused into one single effort per test case ($_sX_{case}$). Any overhead should also be included. Experimental information on back-to-back testing indicates that the ratio between the effort per test case (for mostly non-self-reporting failures) in an environment not using back-to-back testing and the one employing it can be 200 to 400 [Bis86]. In my experience this ratio can be even higher.

In theory, failure of any of the components should register as a difference in version outputs, and should result in a warning from the back-to-back testing harness. The failure will not be detected if the answers from all of the components are identical and wrong. There is also the possibility of receiving a false alarm, i.e. back-to-back testing signals a potential failure where in reality all the answers are correct. The false alarm effect is akin to the "consistent comparison" problem discussed in [Bri87]. The rate at which false alarms occur will depend mainly on the tolerance used for comparisons (unless multiple correct answers are part of the solution). The important thing is that whenever a difference is registered by the testing harness, it counts as a registered failure since it is necessary to inspect the conflicting answers, determine whether a failure has actually occurred, and if it has to proceed with the actions that follow detection of a failure (such as error reporting and failure correction). Therefore an inappropriately tight (small) tolerance, causing a high incidence of false alarms, will increase the effort associated with the failure identification work during back-to-back testing. In this model false alarms are included as a separate item.

The following testing effort model is a generalization of the diversity cost model discussed by Ehrenberger [Ehr85, Sag86]

$$E_{test} = \left(\frac{1}{T_{dev}} \frac{\ln(1-\alpha)}{\ln(1-\varepsilon p_u)} + f_{fa} \right) (1-a_{dev})E + a_{bed}E + a_{pmt}E, \qquad (3.3)$$

where f_{fa} accounts for false alarms, and a_{pmt} describes the post-multiversion testing effort. The cost of false alarms is further expressed as the fraction of test cases, Φ, that has to be investigated, each at an effort which is a fraction f_c of ${}_sX_{case}$. The actual testing and debugging effort is the ratio of the number of equivalent test cases needed to achieve the lower reliability bound $(1-p_u)$ for one version at the α confidence level, and the number (T_{dev}) of equivalent test cases that can be prepared, executed and evaluated during an effort equal to $(1-a_{dev})E$. The value of ε ($0 \le \varepsilon \le 1$) reflects the failure detection (d)efficiency of the testing. The smaller the value of ε the lower is the "visibility" of the faults to the employed testing strategy, and the longer it takes to reach the target failure intensity. In the case of back-to-back testing this parameter may be strongly modified by the presence of correlated faults. Note that E_{test} includes personnel costs for test case preparation, execution and evaluation but not the computer time without human supervision.

The relative effort ratio of the total single version effort (subscript s) to the multiversion effort (subscript bbt) is

$$\frac{{}_sE_{sys}}{{}_{bbt}E_{sys}} = \frac{a_{req} + a_{dev} + \dfrac{1}{{}_sT_{dev}} \dfrac{\ln(1-\alpha)}{\ln(1-{}_s\varepsilon p_u)}(1-a_{dev})}{a_{req1}+ka_{req2}+ka_{dev}+\left(\dfrac{1}{{}_{bbt}T_{dev}} \dfrac{\ln(1-\alpha)}{\ln(1-{}_{bbt}\varepsilon p_u)} + \dfrac{k\Phi f_c}{{}_sT_{dev}} \dfrac{\ln(1-\alpha)}{\ln(1-p_u)}\right)(1-a_{dev})+a_{bed}+a_{pmt}} \qquad (3.4)$$

Note that the total "basic" effort has been factored out of the expression, and that the size of the software, and its complexity, enter only through T_{dev} and a_{dev}. Multiversion failure correlation ($\gamma<1$) enters through the value of ${}_{bbt}\varepsilon$. The break-even point occurs when the relative effort ratio is one. The following examples illustrates the dependence of the ratio on several important parameters.

The effort required to identify a failure is the first indicator of whether back-to-back testing is economic. Self-reporting failures require very limited identification effort and therefore ${}_sX_{case}$ will be about the same or lower than ${}_{bbt}X_{case}$ (remember that ${}_{bbt}X_{case}$ includes fault identification and correction over k components). For self-reporting failures back-to-back testing usually is not cost-effective. We use $\varphi = {}_{bbt}T_{dev}/{}_sT_{dev}$ to denote the relative failure identification and correction ratio.

The influence of failure identification effort is illustrated in Figure 3.1 for an idealized setting. We have assumed that the "basic" testing effort is equivalent to ${}_sT_{dev}=10{,}000$ prepared, executed and evaluated (random and representative) test cases, that the correlation is negligible ${}_s\varepsilon={}_{bbt}\varepsilon=1$ (hence, that post-multiversion testing effort is also negligible, i.e. $a_{pmt}=0$), $a_{req1}=a_{req2}=0.03$, $a_{dev}=0.8$,

α=0.95, Φ=0, and n=2 (i.e. we use a 2-tuple). To be fully effective back-to-back testing requires automated generation of test cases representative of the operational profile, and automated comparison and evaluation of the results. The effort to design such an automated system may be considerable. Sometimes a partial or even full simulation of the environment may be necessary (e.g. for testing of aircraft flight software). In this example we have assumed that this effort is equivalent to that of building another functionally equivalent component, i.e. a_{bed}=1. From the figure we note that as p-->0 the ratio ${}_sEsys/_{bbt}E_{sys}$ --> φ, i.e. in the absence of correlation the failure identification and correction ratio is the governing parameter. It is also obvious that back-to-back is not cost-effective if failures are mostly self-reporting ($\varphi \leq 1$). If we allow post-multiversion testing, then it follows from equation (3.4) that for the multiversion approach to be cost-efficient for small p this effort must remain in the range ${}_sE_{sys}(1 - \frac{1}{\varphi}) > E_{pmt} \geq 0$. This may not be an easy goal. The more faults there are that cannot be detected by back-to-back testing, the larger E_{pmt} needs to be.

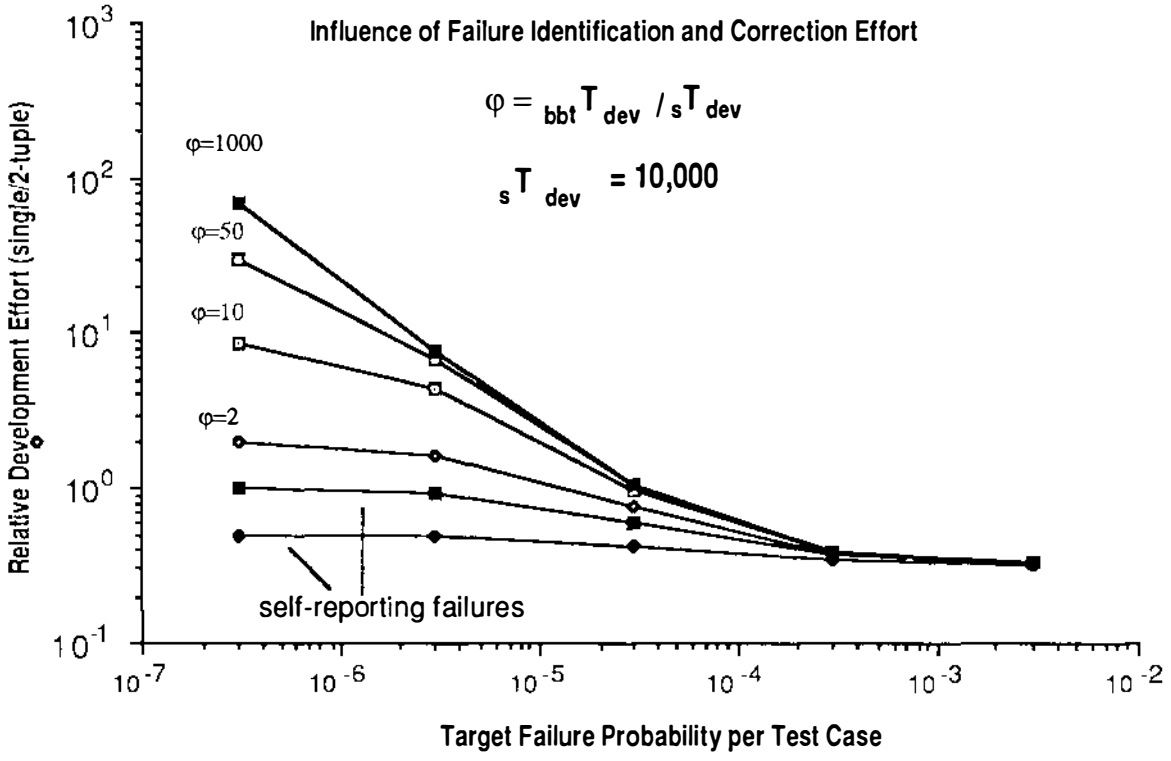


Figure 3.1 Illustration of the influence of the failure identification and correction effort.

There are other effects which can drastically inflate the cost of back-to-back testing. Figure 3.2 illustrates this. In the figure we plot the effort expended in the multiples of the "basic" development

effort E. Unless noted otherwise, all the parameters are the same as for Figure 3.1. The influence of failure correlation is illustrated through the three curves marked φ=10 (note that in the figure $_{bbt}\varepsilon=\varepsilon$). Correlation reduces the "visibility" of similar faults and extends the required testing. For example, correlation of about 90% or higher (corresponds roughly to $\varepsilon \leq 0.1$) may annul any benefits of back-to-back testing.

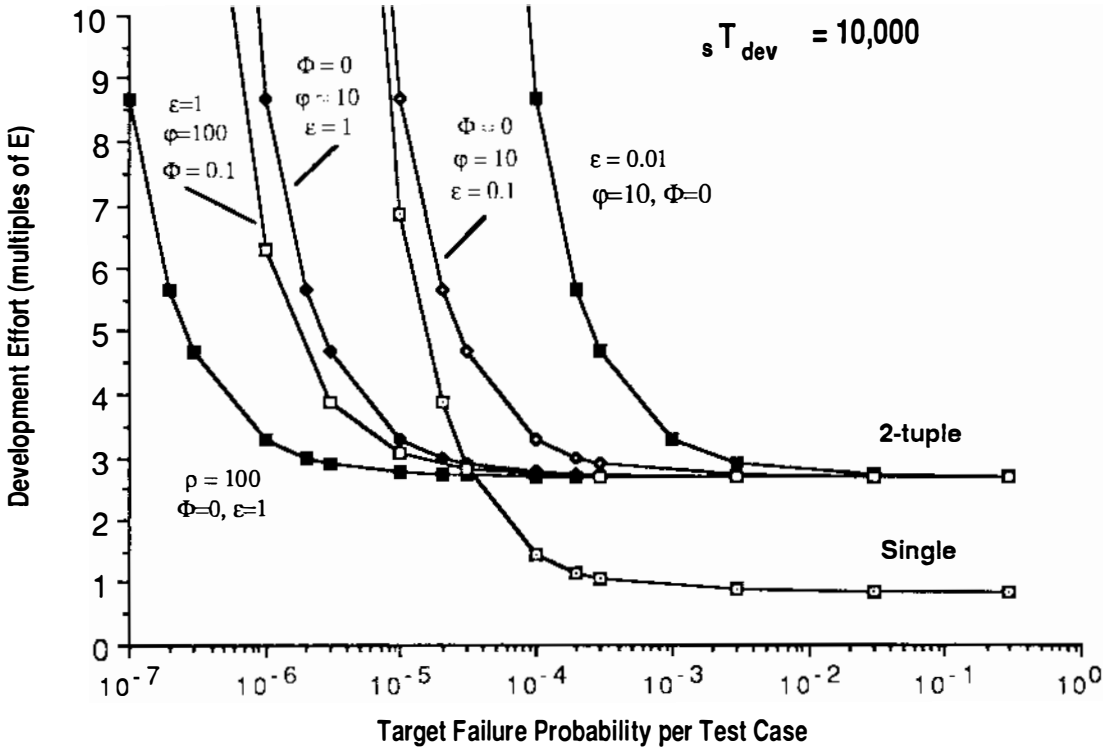


Figure 3.2 Illustration of the cost of diversity in units of the effort needed to develop one hypothetical component vs. the desired system failure probability.

False alarms can be, in principle, as devastating as correlation where the cost is concerned. However, a reasonable false alarm rate can be tolerated (and may even be beneficial, by suffering a few false alarms we may be able to "catch" some more elusive faults much sooner), but high false alarm rates should <u>not</u> be permitted to last throughout the testing. The curves for φ=100, ε=1, Φ=0, and Φ=0.1 (about 10% false alarm rate) with f_c=0.25 (this is very pessimistic) illustrate this.

The advantage of the above model is that it has relatively few parameters and therefore makes the modeling possible on the basis of a few assumptions and values that may be readily available. The disadvantages are several. The controls are rather coarse, and although the model captures the basic

principles of back-to-back testing, it uses a very simple reliability growth model, and may easily produce either overly optimistic projections in favor of back-to-back testing, or overly pessimistic ones. In addition, the model does not account well for situations where $\epsilon=0$ for some faults, although a correction can be made by assigning the single and multiversion processes different target p_u values, and through adjustments of a_{pmt}. In the next section we attempt to remedy some of the problems by using an execution time based reliability growth model to estimate the testing effort.

4. An Execution Time Model

The general model form is the same as in equation (3.1). However, the testing effort is decomposed as follows:

$$E_{test} = E_{FID} + E_{FC} + E_{FA} + E_{UF} + E_{bed} \tag{4.1}$$

E_{FID} is the effort required to identify failures, E_{FC} is the effort required to correct failures, E_{FA} is the effort associated with false alarms, E_{UF} is the additional effort required to remove faults that have not been detected by back-to-back testing (primarily because of failure correlation), and E_{bed} is the cost of the testing harness.

We start by assuming that failure identification effort, both in a single version and in a back-to-back testing environment, is described by the resource effort models proposed by Musa et al. [e.g. Mus87]. The cumulative failure identification effort , ${}_xE_{FID}$, is given by the following expression:

$${}_xE_{FID} = {}_x\vartheta_I \, \tau + a_k \, {}_xX_I \, \mu_x(\tau) \tag{4.2}$$

where subscript x stands for either "s" (single) or "bbt" (multiversion) or "uf" (post-multiversion) testing, ${}_x\vartheta_I$ is the effort per unit of execution time needed to detect that a failure has occurred, τ is the cumulative computer execution time (or some other, suitable, exposure time) expended for program testing, ${}_xX_I$ is the effort per experienced failure spent on failure identification (for example filing of an error report), and $\mu_x(\tau)$ is the expected number of failures by time τ. The value of ${}_{bbt}\vartheta_I$ is expected to be negligible since continuous human supervision of the program outputs is eliminated during back-to-back testing, although a certain small amount of work associated with the process of running the code may still be present. However, the identification work per failure, ${}_{bbt}X_I$, required once a failure has been signalled by the system, will probably be of the same order as in the case of a single program. It may even be slightly larger because, for example, failure reports have to be completed for all involved versions. This is accounted for through a_k, the multiversion failure

identification multiplier ($1 \leq a_k \leq k$).

The function $\mu_{bbt}(\tau)$ describing the expected number of failures by time τ will be more complex since it needs to account for the interaction of two or more components. We model it through an approximation. In the extreme, where a complete mutual exclusion of the program failures is present, the expected number of failures by time τ is the sum of cumulative failures experienced by each component , while in the situation where 100% coincidence of the failures takes place 100% of the time about k times less failures are experienced than if complete phase de-coupling of version failures takes place. To make the analysis tractable we assume that $\mu_{bbt}(\tau) = b_c\, \mu(\tau)$, where $1 \leq b_c \leq k$, is a multiplier that describes the average behavior of the system with respect to the coincidence of the failures from different program versions, and $\mu(\tau)$ describes the expected number of failures of a typical k-tuple version under back-to-back testing.

Failure correction effort can be divided into fault identification and fault correction efforts [Mus87]. We use the following general form

$$ {}_xE_{FC} = k\ {}_xX_{FI}\ \mu_x(\tau) + {}_xX_{FC}\ b_f\ \mu_x(\tau) \tag{4.3} $$

where ${}_xX_{FI}$ is fault identification effort per failure, and ${}_xX_{FC}$ is the fault correction effort per failure. Note that the assumption is that every time a failure has been recorded an attempt is made to find and correct the underlying fault(s), if any. If k versions are being tested, fault identification has to be undertaken for all of them, however fault correction is needed only in those where actual faults are discovered. This is regulated by b_f, the multiplier that describes the average number of faults that have to be corrected for each experienced failure ($b_f \geq 1$). Note that the case of false alarms ($b_f=0$) is treated separately. When giving examples we will assume, for simplicity, that multiversion failures are fully de-coupled ($b_c=k$), and that for every reported failure a fault is corrected in one version ($b_f=1$).

The false alarm effect is modeled by the following equation

$$ E_{FA} = k\ X_{FI}\ \xi_{FA}\ \tau. \tag{4.4} $$

The assumption is that false alarms occur at a rate of ξ_{FA} per unit time, and that the cost associated with them is the effort of identifying a fault (this effort will in most cases be wasted). The fault identification process has to be performed for k versions.

Analysis of similar faults observed in one experiment [Kel88] indicates that these faults have characteristics (such as excitation probability and severity) very like the dissimilar faults. It appears that similar faults differ from the rest of the faults primarily through their invisibility to the failure detection technique used, and an increased failure correction cost. Based on that, we would expect that in a single version environment these faults would behave as dissimilar faults would, and therefore would be detected non-preferentially, with frequency commensurate with their number, throughout the debugging process.

The total development effort for the single version approach is

$$ {}_{s}E_{sys} = (a_{req} + a_{dev})E + {}_{s}\vartheta_I\,\tau + ({}_{s}X_I + {}_{s}X_{FI} + {}_{s}X_{FC})\,\mu_s(\tau), \tag{4.5} $$

where E is, as before, the "basic' development effort and τ_s is the single version testing exposure time. For the multiversion approach the effort is

$$ \begin{aligned} {}_{bbt}E_{sys} = {} & (a_{req1} + k\,a_{req2} + k\,a_{dev} + a_{bcd})\,E + (_{bbt}\vartheta_I + k\,X_{FI}\,\xi_{FA})\,\tau \\ & + (a_k\,{}_{bbt}X_I + k\,{}_{bbt}X_{FI} + {}_{bbt}X_{FC}\,b_f)\,b_c\,\mu(\tau) \\ & + {}_{uf}\vartheta_I\,\tau_{uf} + ({}_{uf}X_I + {}_{uf}X_{FI} + {}_{uf}X_{FC})\,\mu_{uf}(\tau_{uf}) \end{aligned} \tag{4.6} $$

Note that, in general, the debugging time associated with (4.5) will be different from that associated with the multiversion testing part of (4.6), so we distinguish τ_s and τ_{bbt}.

The exact form of the mean value functions will depend on many factors. In the following, we model the reliability growth using the basic execution time model [e.g. Mus87], i.e. we will assume that a general mean value function is described by

$$ \mu_x(\tau_x) = {}_{x}\nu_0\,(1 - e^{-\phi_x B_x \tau_x}) \tag{4.7} $$

where subscript x again stands for either "s", "bbt" or "uf", ${}_{x}\nu_0 = {}_{x}\omega_0/B_x$ is the total expected number of failures, ${}_{x}\omega_0$ is the total expected number of faults remaining in a program at $\tau_x=0$, B_x is the fault reduction factor (faults repaired per failures experienced), and ϕ_x is the per fault hazard rate. If ${}_{s}\omega_0$ is the total number of faults (all are assumed detectable) in the single version, and ${}_{bbt}\omega_0$ is the average number of the faults detectable by back-to-back testing in a typical component of the k-tuple, then we assume ${}_{bbt}\omega_0 + {}_{uf}\omega_0 = {}_{s}\omega_0$. Obviously, back-to-back testing should in practice be supplemented by complementary testing in order to find the faults back-to-back testing may not detect.

The failure intensity function is

$$ {}_x\lambda(\tau_x) = {}_x\lambda_0 \, e^{-\theta_x \tau_x} \tag{4.8} $$

where $\theta_x = \phi_x B_x$, and ${}_x\lambda_0 = {}_x\omega_0 \phi_x$ is the intensity at $\tau_x = 0$. We assume that both testing approaches continue until either some target failure intensity is estimated to have been achieved, or until a target number of failures has been recorded. We also assume that this target intensity (or number of failures) is the same in both cases, i.e. ${}_{bbt}\lambda(\tau_{bbt}) = {}_s\lambda(\tau_s) = \lambda_{target}$, or ${}_{bbt}\mu(\tau_{bbt}) = {}_s\mu(\lambda_{target}) = \mu_{target}$. The first stopping criterion is more pessimistic in the presence of correlated faults because equal failure intensities do no guarantee removal of the same number of faults in the multiversion environment. Most likely a smaller number of failures will be recorded which then requires more extensive post-multiversion testing effort. Hence, in practice the second criterion should be used. If there are faults that are not detected by back-to-back testing, then we will assume that the post-multiversion stopping criterion is given by ${}_{uf}\lambda(\tau_{uf}) = \lambda_{target}$.

Let ${}_{uf}\omega_0$ be the number of residual faults after back-to-back testing. We will assume that the true initial residual failure intensity is ${}_{uf}\lambda_0 = {}_{uf}\omega_0 \phi_s$. A problem with this assumption is that for residual faults of low visibility (e.g. ϕ_s is small if ${}_s\omega_0$ is large), we do not automatically account for the testing needed to establish that. One way of describing the expended effort in situations where ${}_{uf}\lambda_0 \leq \lambda_{target}$ is to assume that the testing time is the time needed to confirm an upper bound on ${}_{uf}\lambda$ by testing for at least $\tau = -\ln(1-\alpha)/\lambda_{target}$ time units, where α is the confidence level. This correction (usually, but not always, small) is not incorporated into the examples.

We use the failure intensity decay parameters (θ_x) to adjust our model for failure correlation which is not 100%, i.e. $\gamma < 1$, by reducing the detection efficiency of back-to-back testing (i.e. $\theta_s \geq \theta_{bbt}$). The inter-version correlation level is incorporated into the model through the modification of the failure intensity decay parameter associated with the versions undergoing back-to-back testing. We simulate the correlation through ratio $r = \theta_{bbt}/\theta_s$. When $r = 1$ there is no correlation, when $r = 0$ correlation is 100%. Faults with $\theta_{bbt} = 0$ are taken out of computations by reducing ${}_{bbt}\omega_0$. Because we have assumed that the cost of failure identification is negligible for back-to-back testing, then so long as $r > 0$, all the failures will eventually be detected. Of course, in practice this may require an inordinate amount of computer execution time which may not be practical from the standpoint of the calendar time schedule and, perhaps, computer time costs. Note that in the current model we are ignoring the cost of computer time. In real life computer time may be a serious limiting factor.

The following examples illustrate the effect of some of the parameters that were not treated

satisfactorily by the simple model presented in section 3. We plot the relative effort ratio ${}_{s}E_{sys}/{}_{bbt}E_{sys}$ against the target failure probability. The starting parameter values used in the examples are similar to the parameters for project T1 in [Mus87].

Unless specified otherwise the parameters have the following values. Failure identification effort is ${}_{s}\vartheta_I={}_{uf}\vartheta_I=9.92$ person-hours per CPU hour. We assume this value for mostly self-reporting failures, in reality a much smaller value is probably more realistic for purely self-reporting failures. The experience gained during a multiversion experiment [Kel88] suggests that truly non-self-reporting failures may require failure identification effort many time the above value. For back-to-back testing ${}_{bbt}\vartheta_I=0$. Also ${}_{s}X_I={}_{bbt}X_I={}_{uf}X_I=0.23$ person-hours per failure. Fault identification effort is ${}_{s}X_{FI} = {}_{bbt}X_{FI} = {}_{uf}X_{FI} = 0.6\ ({}_{s}X_{FI}+{}_{s}X_{FC}) = 0.6\ ({}_{bbt}X_{FI}+{}_{bbt}X_{FC})$ where the factor 0.6 is an average over several published values (see table 5.11 in [Mus87]). The starting failure correction effort is $({}_{s}X_{FI}+{}_{s}X_{FC}) = ({}_{bbt}X_{FI}+{}_{bbt}X_{FC}) = ({}_{uf}X_{FI}+{}_{uf}X_{FC}) = 4.42$ person-hours per failure. The number of faults is ${}_{s}\omega_0={}_{bbt}\omega_0=137$ faults, all assumed detectable by both testing approaches, and initially ${}_{uf}\omega_0=0$. $B_s=B_{bbt}=B_{uf}=1$, $k=2$, $b_c=k$, $b_f=1$, $a_k=k$, $a_{dev}=0.7$, $a_{req1}=a_{req2}=0.03$, $a_{bed}=1$, $\theta_s=\theta_{bbt}=\theta_{uf}=0.156$/CPU hour (no correlation), and $\xi_{FA}=0$ per CPU hour. Note, however that there is experimental evidence that a_{req2} can be as high as 0.20 [Bis86]. Of course, a_{dev} can also vary considerably. Present examples assume a project with "basic" development and testing effort of 5000 person hours for the nominal product reliability (over one CPU hour) of about 0.5.

The fault identification effort still has very strong influence on the effort ratio. The effort ratio is also quite sensitive to the failure intensity decay parameter θ_s. As the the decay parameter gets smaller it takes longer to grow a required reliability. The implication is that if the visibility (θ_s) of the faults may be low under the single version testing strategy, e.g. inexperienced programmers, it may pay to use the multiversion approach, even if failures are almost self-reporting.

We have seen before that the coupling of false alarms with significant inter-component failure correlation can make back-to-back testing very inefficient since the correlation protracts testing time, which permits more false alarms, which in turn makes the cost of failure identification non-negligible and violates the basic premise about multiversion testing. To highlight the behavior of the model in the situation where there is correlation we use a relatively inefficient single version testing process ($D=\theta s=0.0156$/CPU h), and per failure identification effort about 10 times that of nearly self-reporting failures.

The model assumes that the faults that exhibit full correlation over all versions, i.e. all versions return identical and wrong (IAW) answers 100% of the time ($\gamma=1$), are handled through post-multiversion testing. In Figure 4.1 we see that even a small fraction of faults with full correlation

can make the process very uneconomical. In the figure the fraction is denoted by IAW=$_{uf}\omega_0$ / $_s\omega_0$. The stopping criterion used is the pessimistic one of equivalent failure intensities. Because it was assumed that the post-multiversion testing is relatively cheap for low target reliability, but becomes comparable to the single version effort for high target reliability, all the curves with IAW>0 show a maximum. The shaded area represents the region where the model may not be valid because the parameter values may require a completely unrealistic expenditure of the testing effort, an effort that for a version may be many times the "basic" value. It is likely that in practice development would not proceed much beyond the phase where the testing and validation effort is about two or three times the "design and coding" effort, i.e. much over about 70% of the total effort, for each version.

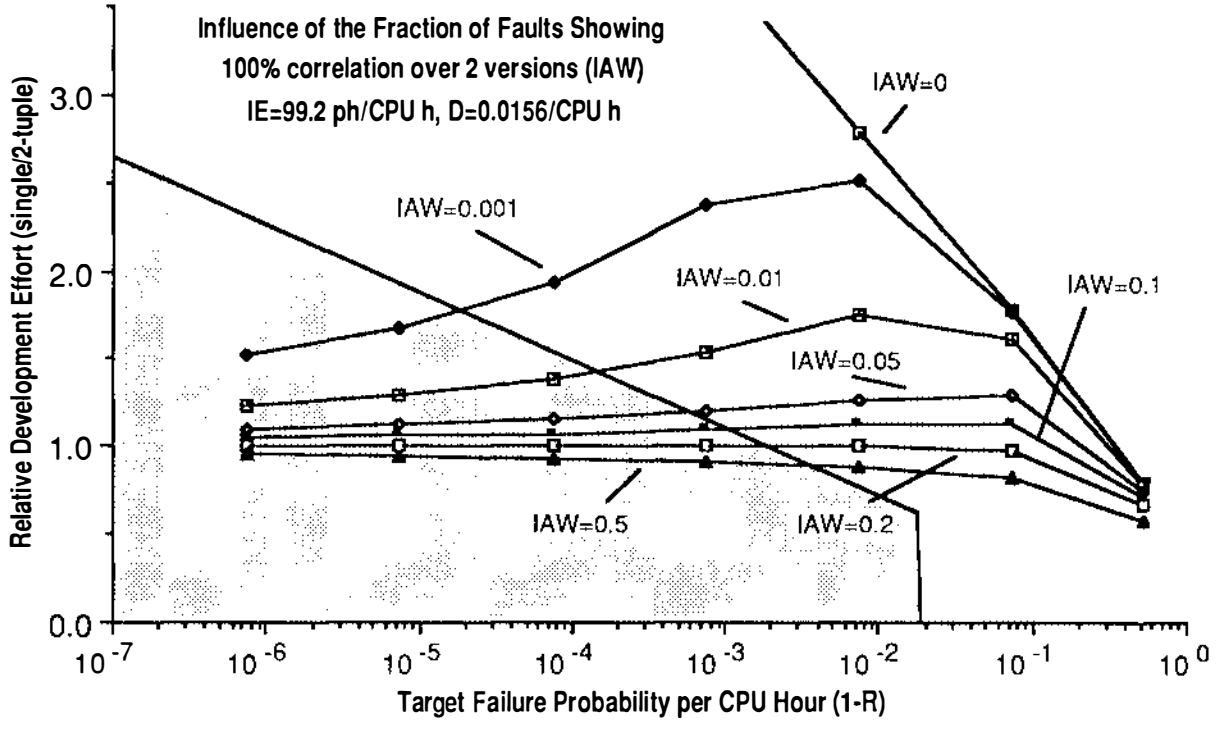


Figure 4.1. Influence of the fraction of faults with 100% correlation across all versions. The fraction is denoted by IAW=$_{uf}\omega_0/_s\omega_0$.

One possible solution for avoiding extensive post-multiversion testing is to develop more than two functionally equivalent versions. The assumption is that this will increase diversity, and perhaps result in no faults with full correlation over all versions. The experimental evidence concerning this assumption is mixed. The maximum spans reported in the literature for identical and wrong

responses stemming from similar faults in diverse software are, for example, zero for 3 developed versions [Vog88], 2 for 6 developed components [Avi87], 3 for 3 [Bis86, Bis88], at least 3 for 8 [Shi88]. It would appear that in most experiments identical and wrong answer span of at least 2 was reported. Variable span was observed for some faults [Kel88]. This suggests that a good starting k-tuple size may be 3. Figure 4.2 illustrates the effect of different k values under no-correlation conditions. Comparison with Figure 4.1 shows that even the k=5 (IAW=0) option may be cheaper than k=2 with IAW over about 5-10%. This indicates that if significant correlation level is suspected it may be better to develop more versions, rather than attempt to find all the residual faults only through post-multiversion testing.

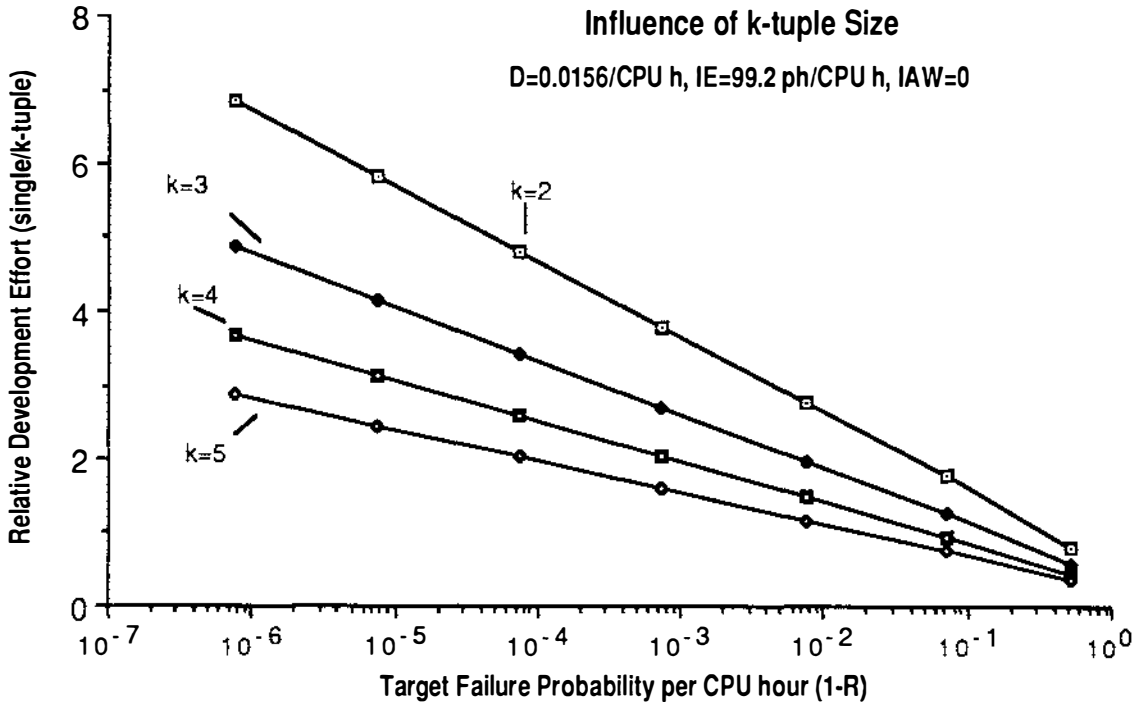


Figure 4.2. Influence of the k-tuple size.

A parameter that may considerably influence the cost-effectiveness is the "basic" effort. It will in practice depend on the target reliability [e.g. Boe81]. Hence, if a project is so large that about 30% of the "basic" effort is sufficient to test the code to the desired reliability, then multiversion approach may be unnecessary. This is illustrated in Figure 4.3 for no-correlation conditions.

The model we have just described is much more sophisticated than the one given in section 3. It accounts for all the major and most of the minor characteristics of back-to-back testing. Almost any

of the empirical multiversion effects can be modeled by adjusting one or more parameters.

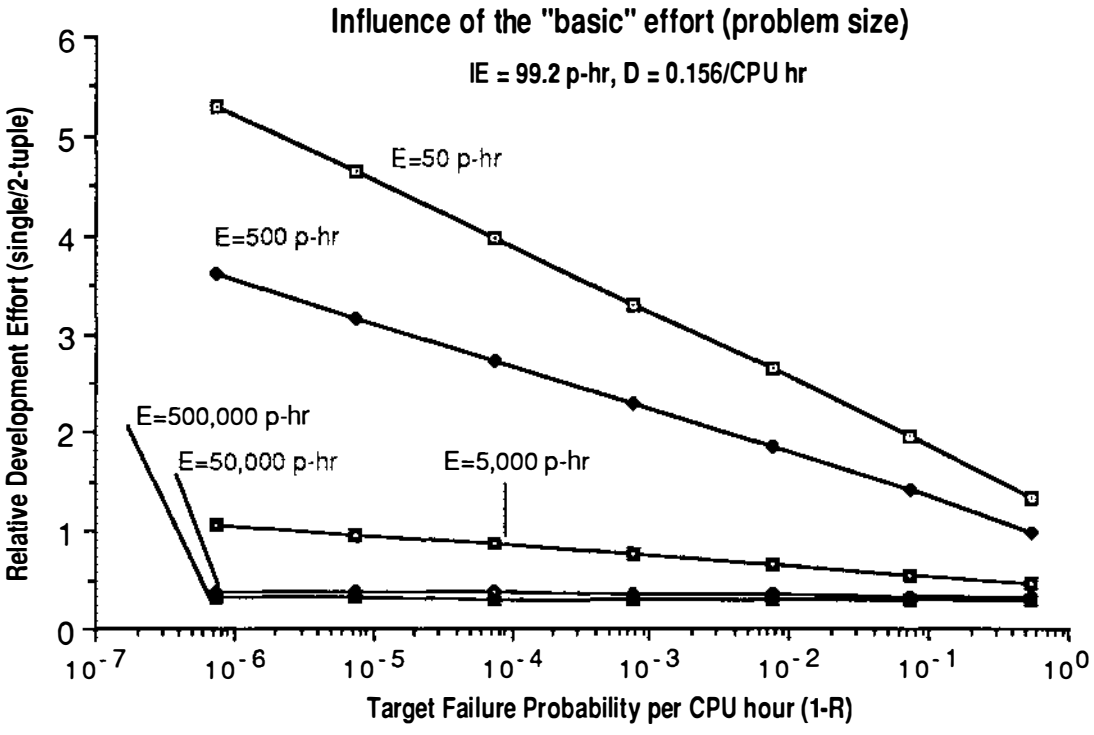


Figure 4.3. Influence of the "basic" effort, E, i.e. problem size.

5. Summary

The published experimental work on back-to-back testing (although sparse and not conclusive) indicates that back-to-back testing can be a very useful. For excample, in [Pan81] it was reported that dual program development (with back-to-back testing) increased the initial development cost by 77% (not factor 2), but in return reduced the number of residual errors from 69 to 2. In the PODS experiment [Bis86], after extensive application of more traditional techniques, nine faults were detected by back-to-back testing, three of which were classified by the authors as "fail-danger" or critical faults. The cost of traditional testing was about 0.4 person-hours per test case, while the cost of back-to-back testing was about 0.16 person-hours per 100 test cases. Similarly, in [Shi88] authors report that 107 faults discovered by back-to-back testing were not detected by any other testing technique they used, while other techniques found about 150 faults not detected by back-to-back testing, etc.

We have described the economics of back-to-back testing using two models. One model is very simple, but allows conceptual understanding of some of the positive properties of back-to-back testing, namely that back-to-back testing can be very cost-effective. The model also highlights the limitations of back-to-back testing, e.g. that it is not economical if failures are mostly self-reporting, and that failure correlation and false alarms can make back-to-back testing far more expensive than single version development. Because of the simplistic reliability growth model used the first cost model is not very stable and tends to be either overly optimistic, in favor of back-to-back testing, or overly pessimistic.

A more detailed and realistic model, based on Musa's resource models and the basic execution time model, was presented next. It accounts well for most of the effects known to occur in a back-to-back testing environment. However, there is still a lot of room for improvement. An extension of the model is currently under investigation. For example, we have not really developed the model in terms of the computer time costs and calendar schedule limitations. Furthermore, if the projected cost is large enough, it may become more economical to use single version program proving, or some other single version validation and verification approach. In principle, the cost-effectiveness of back-to-back testing should always be evaluated in the context of alternatives. The time variable per fault hazard rate and automatic adjustment of the "basic" effort with target reliability may also improve the behavior of the model. The residual fault removal part of the model needs extending to account for confirmation testing in the case of small residual intensity levels. An experimental validation of the model is a priority, but at present it is difficult to make because there are no experimental data that are comprehensive enough. An experiment that will attempt to provide data for model validation is in preparation at NCSU.

In the experience of the author back-to-back testing can be an excellent tool for aiding the development of high reliability software provided it is used appropriately and in conjunction with other validation and verification techniques.

References

[Avi88] A. Avizienis, M.R. Lyu, and W. Schutz, "In Search of Effective Diversity: A six-Language Study of Fault-Tolerant Flight Control Software,", Proc. FTCS 18, pp 15-22, June 1988.

[Bis86] P.G. Bishop, D.G. Esp, M. Barnes, P Humphreys, G. Dahl, and J. Lahti, "PODS--A Project on Diverse Software", IEEE Trans. Soft. Eng., Vol. SE-12(9), 929-940, 1986.

[Bis88] P.G. Bishop, and F.D. Pullen, "PODS Revisited--A Study of Software Failure

Behaviour", Proc. FTCS 18, pp 2-8, June 1988.

[Boe81] B.W. Boehm, *Software Engineering Economics,* Prentice-Hall, Inc, Englewood Cliffs, N.J. , 1981.

[BrK86] S. Brilliant and J.C. Knight, "Testing Software Using Multiple Versions", University of Virginia, Department of Computer Science, Report No. RM-86-07, 1986.

[Bri87] S. Brilliant, J.C. Knight, and N.G. Leveson, "The Consistent Comparison Problem in N-Version Software," ACM SIGSOFT Software Engineering Notes, Vol. 12 (1), pp 29-34, 1987.

[Dur84] J.W. Duran and S.C. Ntafos, "An evaluation of random testing", IEEE Trans. Soft. Eng., Vol. SE-10, 438-444, 1984

[Eck85] D.E. Eckhardt, Jr. and L.D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors", IEEE Trans. Soft. Eng., Vol. SE-11(12), 1511-1517, 1985.

[Ehr85] W. Ehrenberger, "Statistical Testing of Real Time Software", in "Verification and Validation of Real Time Software", ed. W.J. Quirk, Springer-Verlag, 147-178, 1985.

[Fis75] M.A. Fishler et al., "Distinct Software: An Approach to Reliable Computing," Proc., 1975 USA-Japan Computer Conf.

[Gil77] T. Gilb, *Software Metrics*, Winthrop Publishers Inc., Cambridge, Massachusets, 1977.

[Kel88] J. Kelly, D. Eckhardt, A. Caglayan, J. Knight, D. McAllister, M. Vouk, "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results", Proc. FTCS 18, pp 9-14, June 1988.

[Kni86] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the assumption of Independence in Multiversion Programming", IEEE Trans. Soft. Eng., Vol. SE-12(1), 96-109, 1986.

[Lit87] B. Littlewood, and D.R. Miller, "A Conceptual Model of Multi-Version Software," FTCS 17, Digest of Papers, IEEE Comp. Soc. Press, pp 150-155, July 1987.

[Mus87] J. Musa, A. Iannino, and K. Okumoto, "Software Reliability: Measurement, Prediction, Application," McGraw-Hill Book Co., 1987.

[Pan81] D.J. Panzl, " A Method for Evaluating Software Development Techniques", The Journal of Systems Software, Vol. 2, 133-137, 1981.

[Pro88] P.W. Protzel, "Automatically Generated Acceptance Test: A Software Reliability Experiment," 2nd Workshop on Software Testing, Verification and Analysis, Banff, IEEE Comp. Soc., pp 196-203, July 1988.

[Ram81] C.V. Ramamoorthy, Y.K.R. Mok, F.B. Bastani, G.H. Chin and K. Suzuki, " Application of a Methodology for the development and validation of reliable process control software," IEEE Trans. Soft. Eng., Vol. SE-7 (6), 537-555, 1981.

[Sag86] F. Saglietti and W. Ehrenberger, "Software Diversity -- Some Considerations about Benefits and its Limitations", Proc. IFAC SAFECOMP '86, 27-34, 1986.

[Sco84] R.K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs, "Investigating Version Dependence in Fault-Tolerant Software", AGARD 361, pp. 21.1-21.10, 1984 .

[Shi88] T.J. Shimeall and N.G. Leveson, "An Empirical Comparison of Software Fault-Tolerance and Fault Elimination," 2nd Workshop on Software Testing, Verification and Analysis, Banff, IEEE Comp. Soc., pp 180-187, July 1988.

[Vog88] U. Voges, "Use of Diversity in Experimental Reactor Safety Systems," in *Software Diversity in Computerized Control Systems*, U. Voges (ed.), Springer-Verlag, Wien, Austria, pp 29-49, 1988.

[Vou85] M.A. Vouk, D.F. McAllister, K.C. Tai, "Identification of correlated failures of fault-tolerant software systems", in Proc. COMPSAC 85, 437-444, 1985.

[Vou88] M.A. Vouk, "On Back-To-Back Testing," Proc. COMPASS '88, pp 84-91, June 1988.

An Execution Backtracking Approach to Program Debugging

Hiralal Agrawal

Eugene H. Spafford

Software Engineering Research Center
Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907-2004
(317) 494-7825

ha@cs.purdue.edu
spaf@cs.purdue.edu

Abstract

An execution backtracking facility in interactive source debuggers could support the thought process some users employ while debugging — working backwards from the location where an error is manifested and determining the conditions under which the error could occur. Such a facility would also allow a user to change program characteristics and reexecute from arbitrary points within the program under examination — a ''what-if'' capability.

This paper describes work in progress to develop a debugger with such a backtracking function. We describe why the facility is useful, and why other current techniques are inadequate. We show how execution backtracking can be efficiently implemented by saving only the latest values of variables modified by a statement, and allowing backtracking only over *complete* program statements.

Biographical Sketches

Hiralal Agrawal received a M.Sc.(Tech.) degree in Computer Science from Birla Institute of Technology and Science, Pilani, India in 1985. He is currently a Ph.D. candidate in the Department of Computer Sciences at Purdue University. His research interests are software testing and debugging, programming languages, and distributed systems. He received a Purdue University Fellowship from 1986 to 1988. He is a member of the Association for the Computing Machinery, IEEE and the IEEE Computer Society.

Gene Spafford received a Ph.D. in Information and Computer Science from Georgia Institute of Technology in 1986 for his design and implementation of the Clouds distributed operating system kernel. He next spent 18 months in a post-doc research scientist position with the Software Engineering Center at Georgia Tech. In August 1987, Gene joined the faculty of the Department of Computer Sciences at Purdue University as an assistant professor. He is also an active researcher with the NSF/Purdue/University of Florida Software Engineering Research Center (SERC). Besides a well-known interest in the Usenet and other forms of electronic conferencing, Spaf works with distributed systems, computer security, software testing and debugging, and professional ethics issues. He is a member of ACM, IEEE and IEEE-CS, MAA, and Usenix.

An Execution Backtracking Approach to Program Debugging

Hiralal Agrawal
Eugene H. Spafford

Software Engineering Research Center
Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907-2004

1. Introduction

The importance of good debugging tools cannot be overemphasized. Average programmers may spend considerable amounts (possibly more than 50%) of their program development time debugging. Several tools are available to help them in this task,[1] varying from hexadecimal dumps of program state at the time of failure to window and mouse-based interactive debuggers using bit-mapped displays [Adam86, Carg86]. Most interactive debuggers provide breakpoints and traces as their main debugging aids [Kats79, Mara79, Dunl86]. The work described in this paper involves the development of an execution backtracking facility to be used in association with breakpoints and traces as an aid in debugging. We believe that such a facility will be a significant addition to the debugging help provided by conventional debuggers today.

Current Approaches to Interactive Debugging

The simplest and perhaps the most common technique for debugging involves the insertion of *print* statements in the program to display desired control-flow information and intermediate values of certain variables. This approach can be tailored for the problem at hand and, as such, can be quite effective. However, this approach can also be tedious to use because program source has to be modified and recompiled every time some new information is desired.

Interactive debuggers allow run-time control over display of debugging information. The program source does not have to be modified and recompiled to display desired information. These interactive debuggers provide *breakpoints* as the main facility for debugging. The user specifies one or more breakpoints, and whenever the control reaches any of these locations the program execution is suspended and control is passed to the debugger. The user can then inspect the program state by displaying current values of variables. Some debuggers also allow the control stack and saved contexts to be displayed. A few also allow program state to be explicitly modified by the user during execution. After examining (and possibly changing) values, the user can set new breakpoints and continue execution.

[1] See, for instance, [Agra88a] and [McDo87].

Interactive debuggers also allow *tracing* information to be specified. The user can specify trace points and corresponding information to be displayed at these points. Trace information can be simple messages about control reaching the trace-points, or it can be values of certain variables at these points. Whenever control reaches any of the trace points, the specified trace information is displayed and execution continues. Note that a trace point is a special case of a breakpoint where the specified trace information is displayed and execution continued automatically.

Some debuggers also allow certain conditions about program state to be associated with trace points. In these systems, whenever a trace point is reached, the corresponding specified condition is automatically evaluated. If the condition is met then the specified trace information for that point is displayed. Otherwise the execution continues without displaying any information. Similarly, conditions can also be associated with breakpoints in such debuggers.

Another facility provided by some interactive debuggers allows the user to *single-step* through a program. The program is executed one statement at a time and control returned to the user after every statement. Note that single-stepping is also a special case of breakpoints where breakpoints are set after consecutive source statements.

Thus, we can see that the major form of debugging support commonly available appears to be some form of breakpoint mechanism coupled with a display facility. All the common debuggers with which we are familiar provide these functions, and all require the user to debug in a sequential "forward" manner.

How Does One Debug?

Given that a program has failed to produce the desired output, how does one go about finding where it went wrong? Other than the program source, the only important information available to the programmer is the input data (if any), and the erroneous output produced by the program. If the program is sufficiently small, it can be analyzed manually on the given input. However, for large programs, such analysis is much too difficult to perform. One logical way to proceed in such situations would be to *think backwards* — deduce the conditions under which the program produces the (incorrect) output that it did [Schw71, Goul75, Luke80].

Conceptually, this *backwards execution* can be done by analyzing the effect of each source statement, starting at the point the error is manifested, and proceeding backwards one statement at a time. For large programs such analysis at the statement level would be too tedious to perform. What we seek is to first narrow the user's focus to a small region of the program that is likely to contain an error, and then do statement level analysis within this region.

How can a user determine the small program region that is most likely to contain an error? If we consider small logical regions of the program as "logically atomic"[2] blocks, then this problem is analogous to locating an erroneous statement in the program. Only now the program is viewed as a sequence of blocks instead of individual statements; one need only to think backwards at the program block level instead of the statement level.

To do such backward analysis using conventional interactive debuggers, one would first set a breakpoint just before the last logical block. If the program state is found to be correct at this point, it would imply that the error occurred within the last block. Otherwise another

[2] We mean *indivisible* here as opposed to any intimation of fault tolerance.

breakpoint would be set before the second-last block, and the program *reexecuted.* If the program state is found to be correct at that point, then we conclude that the error resides within that second-last block. Otherwise this process of setting breakpoints in backward order and reexecuting the program continues until the erroneous block is discovered. Remember that we are concerned with debugging large programs where any forward analysis might be very time consuming.

If N is the number of blocks in the program, then clearly this method of setting breakpoints successively in backwards order and reexecuting the program every time leads to an $O(N^2)$ execution behavior. This is because, for the successive reexecutions of the program up to the breakpoints set successively in backward direction, it will take N, $N-1$, $N-2$, ... $N-i$ block executions if the error is found in the i'th block from the end. So it will take a total of $N + (N-1) + (N-2) + ... + (N-i)$ block executions. In the worst case, $i=N$, and this will require $\frac{N\times(N+1)}{2}$ block executions, thus the $O(N^2)$ behavior.

The scenario presented above is admittedly an over-simplification and over-mechanization of real-life situations encountered while debugging programs. Most of the time, much more information about the program *behavior*, internal and external, is known to the programmers. With that information, the programmers do not always have to follow the rigid path outlined above. However, in large and complicated programs, especially those developed by teams of programmers and maintained by people who were not involved with the code development, strategies somewhat similar to the one mentioned above must be employed to detect the lurking bugs most frustrating in real systems development.

Why Backtrack While Debugging?

If interactive debuggers provided a facility to backtrack the program execution, then there would be no need to restart the program execution every time a new breakpoint is set prior to the current one. Using such a backtracking facility, the user could backtrack the program execution to any earlier desired location. If the control path followed by the program in its initial execution contained the specified location, the debugger would restore the program state to that when control last reached that location. Otherwise, the user would be informed of an error.

Using such a backtracking facility, the backward debugging strategy discussed above would require only $O(N)$ block executions. This is because, from the point an error is manifested, program execution could be backtracked one block at a time, checking for errors in program state after each backtrack. The program would not have to be repeatedly reexecuted from the beginning for each block under examination.

Even if the user does not use a backward debugging strategy, an execution backtracking facility would still be useful. When debugging using the conventional interactive debuggers, the user has to determine the regions that are likely to contain errors and then set breakpoints in those regions. Often, when the program execution is suspended at a breakpoint, the user discovers that the error occurred at an earlier location. In such a case, there is no other choice but to set another breakpoint at an earlier location and start the program execution again; this may require multiple iterations before enough context is gained to identify the bug. With an execution backtracking facility, program execution could simply be restarted at any desired earlier location, and there would be no need to rerun the entire program.

As yet another motivating example, consider a user single-stepping through a program, observing its behavior or tracking down a bug. The user may "step over" a statement by mistake or by not realizing its importance until some statements later in the execution. In the absence of a backtracking facility, the only way to recover from this situation is to restart the program execution and take care that the same mistake is not made again. With an execution backtracking facility, any single-step command could easily be undone by simply backtracking the execution over that single statement.

If the debugger also allows the user to modify the program state, then the execution backtracking facility could be used to do some *what-if* analysis over sections of the program. Starting from a particular program state, the user could execute a section of the program, inspect the results of this execution, backtrack the execution to the same earlier state, change this state, and reexecute over that program section. Different execution paths taken by the program could be easily examined using this backtracking and state changing facility.

From these examples, we can see that having some form of backtracking function could be quite useful in a debugger. At the least, users will feel more comfortable and confident if they know they can undo their actions [Hans71, Bran83]. At best, the presence of such a facility could make a significant difference in the time spent debugging a large amount of code.

2. Related Work

The concept of reverting state in a system is not new. We describe some significant related ideas here, and explain why they are different or inappropriate for the approach we have just outlined.

Checkpointing

Database systems have long been using a concept similar to execution backtracking to implement recovery from system failures [Verh78, Haer83]. The approach normally used involves some form of periodic *checkpointing* of the database, and maintaining a transaction log. When a failure occurs that leaves the database in an inconsistent state, the database is first restored to the state during the last checkpoint, and then the transaction log is replayed against the restored database.

One difference between the rollback recovery in databases and our view of execution backtracking in debugging is in the *granularity of backtracking*. This granularity is coarse for database rollback recovery because the checkpoints are usually far apart in time. This is because the purpose of database backtracking is recovery from failures, which are expected to be infrequent. But the granularity of backtracking required for debugging is fine because it needs to be performed at the source statement level.

Furthermore, for databases the transaction log and state information recorded before a recent consistent state can be discarded as long as the system can be backed up to this state. This is sufficient because the mechanism is meant for recovery to *a* consistent state, and not to any arbitrary point in the history of the database. To support backtracking for debugging as we have described it would require the ability to go back to any point arbitrarily far back in the history of the execution, thus requiring an arbitrarily long log file.

Recovery Blocks

Another form of execution backtracking arises in fault-tolerant software systems using the recovery-block technique [Rand75]. In this approach to fault-tolerance, several alternate algorithms are used to perform a single task. First, one algorithm is executed and its results are evaluated using an acceptance test. If the test fails, then the program state is restored to that at the beginning of this algorithm, and an alternate algorithm is then executed. This process — executing an algorithm, performing an acceptance test on its results, then rolling back the program state if the test fails and executing another alternate algorithm — continues until either an acceptance test succeeds, in which case the task succeeds, or until all the alternate algorithms have failed, in which case the task fails.

One difference between backtracking in recovery-block techniques and that in debugging is that, in the former, the granularity of backtracking is fixed at the algorithm level and it is not user-controlled at run-time. Unlike our proposed use in a debugger, the number of statements rolled-back is programmed into the recovery-block code and is not under user control.

Zelkowitz's RETRACE

Zelkowitz incorporated a backtracking facility within the programming language PL/1 by adding a RETRACE statement to the language [Zelk73, Zelk71]. With this statement, execution could be backtracked over a desired number of statements, up to a statement with a given label, or until the program state matched a certain condition. Implementation of RETRACE required maintaining an in-core trace table of assignments to variables and control flow information.

This incorporation of backtracking facilities within a programming language can be useful in programming applications where several alternate paths should be tried to reach a goal. Such problems frequently arise in artificial intelligence applications, for instance. However, this approach is somewhat similar to the recover-block approach in that the user must program the RETRACE statements into the code. Though useful, it does not provide an *interactive* control over backtracking while debugging.

EXDAMS

EXDAMS, an interactive debugging tool developed in 1969, also provides an execution replay facility [Balz69]. In EXDAMS, the program to be debugged is first executed and a history containing a trace of complete control-flow and all state-changes is built. This is done by recording the direction followed in the **if-then-else** statements and at the end of **do** loops. The values of variables on the left hand side of assignment statements are saved. Once the complete history is built, the program is "executed" through a "playback" of this tape. At any point, the program execution could be backtracked to an earlier location using the information saved on the history tape.

A serious problem with this approach of saving the complete history of control-flow and state changes is that it could take enormously large space when debugging long-running programs. The space required to save such histories would also be a function of the program input, and thus cannot, in general, be bounded in advance. Furthermore, after backtracking to an arbitrary point in the program, the user cannot change values of variables before executing forwards again from there, because EXDAMS simply replays the program behavior recorded earlier.

COPE

COPE, an integrated programming environment, also provides mechanisms for backtracking at command level while editing or executing programs [Arch84]. In COPE, all operations are reflected as changes in its file system. Whenever the contents of any file block change, a new copy of that block is saved. The old copy is not overwritten. The block numbers of both the old and the new blocks are recorded in a command log maintained separately. To undo a command, the old file blocks simply replace the new ones.

This approach, when applied to statement-level backtracking, suffers from a severe space problem: Even if a statement execution results in a small change in a file block, the whole block is saved again. Furthermore, a continually growing log is required.

INTERLISP

The INTERLISP system, a program development environment for the LISP language, also provides recovery facilities within the language framework [Teit72, Teit78]. It provides UNDO and REDO functions whose implementations are embedded within the language processor. INTERLISP also maintains a history-list of all commands where it records the side effects of all operations. Operations can then be backtracked using the information saved in the history-list.

Saving the complete history of side effects leads to the same space problems mentioned above for EXDAMS. To overcome this problem, a bounded history-list is used: after it grows to a predetermined limit, old events are forgotten as new events occur. This is obviously not desirable under our approach, as backtracking to points arbitrarily far back may be required during debugging.

Also, INTERLISP provides a functional programming environment, whereas our interest here is to provide a backtracking facility for debugging in the more common procedural environment.

3. Debugging with Structured Backtracking

Most of the systems discussed above used some form of saving the control flow history and the state changes to implement execution backtracking. As mentioned earlier, this approach has a major drawback: such records may take arbitrarily large amounts of space while debugging long-running programs, and the space required cannot, in general, be bounded in advance. In this section, we will describe a different approach to implementing execution backtracking that does not have this drawback. Under our approach, we only save the latest values of the variables changed by any statement, and we restrict backtracking to be over *complete* statements only.[3] Under this approach, the space required is proportional to the size of the program (we develop this below), and can be bounded (and thus allocated) at compile time.

Each program statement has a *change-set* associated with it. The change-set of a statement consists of all those variables whose values *could* be modified by that statement. For example, the change-set of an assignment statement would be a singleton consisting of the variable on

[3] Our definition of a complete statement may be thought of as equivalent to any statement in Pascal up to its terminating semicolon — simple assignment statements, *while* loops and procedure invocations with arbitrary expression as parameters are all *complete* statements.

its left hand side. The change-set of a read statement would be the set of all variables whose values are read by the statement. The change-set of a procedure call would be the set of all those parameters and global variables whose values could be modified by the procedure invocation. Henceforth, we refer to all statements that assign new values to variables as *assignment statements.*

Change-sets of composite statements are computed using the change-sets of their constituent assignment statements. If we denote the function computing the change-set of a statement by C, source statements by S, S_1, and S_2, arithmetic expressions by exp_1 and exp_2, and a boolean expression by *cond*, then the change-sets of some composite program constructs are computed as follows:[4]

$$C(S_1;S_2) = C(S_1) \cup C(S_2)$$

$$C(\textbf{ if } cond \textbf{ then } S) = C(S)$$

$$C(\textbf{ if } cond \textbf{ then } S_1 \textbf{ else } S_2) = C(S_1) \cup C(S_2)$$

$$C(\textbf{ while } cond \textbf{ do } S) = C(S)$$

$$C(\textbf{ repeat } S \textbf{ until } cond) = C(S)$$

$$C(\textbf{ for } index\text{-}var := exp_1 \textbf{ to } exp_2 \textbf{ do } S) = C(S) \cup \{index\text{-}var\}$$

Change-sets of other composite statements can be computed in a similar manner.

For each complete statement, the debugger allocates space to save just one instance of the values of all variables in its change-set. Just before executing any statement, it saves the values of all variables in the change-set of that statement in the space allocated for this purpose. After that, the statement is executed. Later, if the execution has to be backtracked over this statement, the values of variables in the change-set of this statement are simply restored.[5]

Under this approach, backtracking can be done only over complete statements. Backtracking from a statement outside the body of a composite statement to a statement inside it is not allowed. For example, one cannot backtrack from a statement outside a loop to a statement inside it. If one has to backtrack to the middle of a complex statement (like a loop) from a location outside it, one should backtrack first to the beginning of the statement, and then forward execute up to the desired location inside it.[6] From a statement inside a loop body, one can backtrack to an earlier statement in the loop body for the same loop iteration, but inter-iteration backtracking would not be possible.

The restriction on backtracking over only complete statements is not an unduly constraining one. In a sense, it is similar to encouraging structured execution in the backward direction. As such, analyzing the effects of statements in reverse order should be much easier and logical

[4] For simplicity, we assume that all expressions are side-effect-free. The approach discussed here can easily be generalized to include expressions with side-effects.

[5] This scheme is superficially similar to a database change log mechanism. However, in our scheme, we do not log every change to every variable.

[6] If desired, this process can also be automated by the debugger.

since the user needs to consider only one complete statement at a time.

Note that the use of change-set restoration puts all involved variables back into their prior state. It is then possible to execute forward from that point, possibly after the user changes some data values. This provides the *what-if* capability referred to earlier.

To illustrate how backtracking is constrained, consider the following program segment:

```
S1:    ....
S2:    while cond do begin
S3:        ....
S4:        ....
S5:        ....
       end;
S6:    ....
S7:    if cond then begin
S8:        ....
S9:        ....
       end else begin
S10:       ....
S11:       ....
       end;
S12:   ....
S13:   ....
```

All the following instances of backtracking are *not* allowed under our scheme:

from S6 to S5
from S12 to S9
from S9 to S3
from S3 in iteration i to S5 in iteration $i-1$

Following are the examples of some valid instances of backtracking:

from S2 to S1
from S5 to S3 within the same iteration
from S6 to S2
from S4 to S1
from S9 to S8
from S13 to S7

Note that one can backtrack from a statement inside a loop to a statement outside it. These restrictions are just like those followed in structured programming and included in most modern language standards — disallowing *goto* statements that jump inside a loop from outside, but allowing *breaks* from inside a loop to outside.

Recursion is handled in the same way as iteration. From a statement inside a recursive procedure, backtracking to an earlier statement in the same procedure is allowed only within the current invocation of that procedure; inter-invocation backtracking, like inter-iteration backtracking, is not allowed.

Presence of goto statements causes special problems that cannot be handled easily under our approach. In most modern programming languages, gotos are useful mainly in situations

where, on detecting some special or erroneous condition, control must jump from a statement nested several levels deep, to a statement some levels outside. In these situations, backtracking would imply jumping from outside a composite statement to inside it, which is not allowed in our approach. We believe such situations are infrequent in most programs, and when they do occur, users could work their way around by backtracking to an earlier statement and then executing forwards from there.

Under the structured backtracking approach, we only need to save the latest instance of values of variables in the change-sets of all statements. This is unlike history traces where one has to save all instances of values taken by all variables. For example, consider the following program segment to compute 2^N, $N \geq 0$:

```
S1:   power := 1;
S2:   i := 1;
S3:   while (i <= N) do begin
S4:       power := power * 2;
S5:       i := i + 1;
      end;
```

The change-sets for these statements are as follows:

```
S1:   { power }
S2:   { i }
S3:   { power, i }
S4:   { power }
S5:   { i }
```

In the history-trace approach, *all* values taken by variables *power* and *i* in statements S4 and S5 would be saved. But in our approach, only one instance each for the two statements they belong to, would be saved: in the change-set of statement S3, and in the change-sets of statements S4 and S5.

For the above example, the space required to save a complete history trace depends on the value of N. If the value of N is read at run-time, then the space required cannot be bounded in advance. On the other hand, in our scheme, the space required depends only on the sum of the sizes of change-sets of all statements, and thus can be easily computed at compile time. For the above example, we need only six words of space for *any* value of N because the sum of the sizes of change-sets of all the statements is six. The history trace approach would require $2\times(N+1)$ locations for the above example.

Handling Pointers

How do we compute the change set of an assignment statement if the value is assigned indirectly through a pointer? Most data-flow analysis algorithms, in similar situations, take a conservative approach — it is assumed that an indirect assignment through a pointer could potentially modify any variable. This is because at compile time, we do not know all variables the pointer might reference at run time. However, this does not pose a major problem for handling execution backtracking in our scheme. Consider, for example, the following Pascal indirect assignment statement:

```
int_pointer↑ := num;
```

At compile time, we do not really know what the change-set of this statement might be. Fortunately, we do not really need to know the *name* of variables in the change-set of this statement. At compile time we only need to allocate enough space to save information to enable execution backtracking at run time. For the change-set of the above statement, we only need to allocate two words — one to save the *value* of the variable being assigned a value, and one to record the *address* of that variable. At run time, just before executing this statement, the debugger would save both the address as well as the value of that variable before executing that statement. When the execution has to be backtracked over this statement, the value saved would be restored at the address recorded.

When indirect assignment statements appear within composite statements it becomes more difficult to calculate the change-set. For example, if there is an indirect assignment in a loop body, the l-value of the assignment could change from one iteration to another. In such cases, we have to save all the address-value pairs assigned in the loop at run-time. The number of such pairs may not be easily calculated before execution; we are continuing to investigate efficient ways to handle pointers in composite statements.

Other Special Cases

We are still examining the question of how to backtrack over I/O operations. Any system can at most undo things directly under its control. If any of its actions has effects outside the boundary of the system, then the system, in general, cannot always retract them. For instance, we have no way to save the ''state'' of a tape drive or line-printer to allow us to back up and restart it from there. We are currently examining the semantics of backtracking in the presence of I/O, and the possible approaches to handle it. One possible approach involves the use of I/O buffering with *pushback* operations, similar to the ''ungetc'' operation in the C programming language standard library.

Furthermore, the problems of asynchronous signals and IPC are still unaddressed by our research. There is some promise in the notion of *virtual time* as discussed by Schiffenbauer [Schi81] and employed in distributed monitors such as RADAR [Robb83], but we have not yet examined this area in depth.

4. Space Requirements

If an assignment statement is nested N levels deep, then the variables modified by it would belong to change-sets of N statements — $N-1$ composite statements in which it is nested, and the assignment statement itself.[7] Let A be the total number of assignment statements in the program, s_i be the size of change set of the ith assignment statement (remember that read statements or procedure calls are also referred to as assignments), n_i be the nesting level of the ith assignment statement, and S be the sum of sizes of change-sets of all statements in the program. Then we have:

$$S = \sum_{i=1}^{A} (n_i \times s_i)$$

[7] For simplicity, an assignment statement not nested in any composite statement is assumed to be at level one; an assignment statement with a single enclosing *if* or *while* is assumed to be at level two; and so on.

Or,

$$S \leq A \times \max_{i=1}^{A} \{n_i\} \times \max_{i=1}^{A} \{s_i\} \quad (1)$$

Let

$$\alpha = \max_{i=1}^{A} \{n_i\} \quad (2)$$

and

$$\beta = \max_{i=1}^{A} \{s_i\} \quad (3)$$

That is, α represents the maximum nesting level in the program, and β represents the size of the largest change set of all assignment statements in the program.

Let L be the length of the program in number of source lines. The number of assignment statements in the program is bounded by the program length, so we also have:

$$A \leq L \quad (4)$$

Then, from (1), (2), (3), and (4), we have:

$$S \leq L \times \alpha \times \beta \quad (5)$$

In practice, both α and β are usually small constants, depending on the program being debugged. If we denote the product $\alpha \times \beta$ by c, we get from (5):

$$S \leq c \times L$$

Or,

$$S = O(L)$$

That is, normally, the sum of the sizes of the change-sets of all statements in the program is of the order of the length of the program. Thus, the expected (average) space required to keep backtracking information is proportional to the program length.[8]

5. Implementation and Extensions

Implementation of our method in an existing system should be straightforward. The support for saving state can be built into the code generator portion of the compiler. All that is necessary is to recognize the change-set for each statement, and generate additional storage and store instructions per statement. Some additional marking code would be required to save active display registers and stack frames so that the proper register context could be restored during a backtrack. This is not a complicated task and it could be done as part of the prelude/postlude code in each subroutine.

Support within the debugger interface should likewise be easy to provide. All that is required is to identify where to backtrack in the program under examination, then restore the necessary context.

---

[8] The worst case upper bound is $O(L^2)$, and the reader is referred to [Agra88b] for the complete derivation.

It may be noted that it is possible, with our method, to save the last *N* (for arbitrary *N*) values of each variable instead of just the last instance. Sometimes, this will allow the user to look at the values produced in the last few iterations of a loop, or the last few invocations of a function. The storage required to save these additional values is still bounded, and the additional storage and computation overhead will still be (on average) linear in the size of the program.

Another simple extension is the notion of saving change-sets for only selected portions of code. We envision this selection feature as being most useful when debugging code consisting partially of well-tested functions such as might be in a library or reused code from another project. The well-tested code may not require any saving of change-sets, or saving only the single most recent value of variables, while new code may be instrumented to save the last 5 or 10 values. This is analogous to linking a library compiled without debugging support into a program being debugged.

6. Related Ideas

We intend to investigate the possibilities of *reverse execution* of code that has not been checkpointing its state, rather than simply *backtracking* the execution as discussed in this paper. Such a facility would be useful in cases where programs have not been compiled with debugging support enabled, and where an unexpected software failure has occurred. This approach requires non-deterministic execution because of the presence of uninvertible assignment statements in the program. Our initial investigations have lead us to believe that we can provide some valuable debugging help with such a mechanism.

Consider, for example, the following code fragment:

```
if ( k < 0 ) then
   i := -1
else
   i := 2;
j := sqrt (m);
u := j * i;
i := 0;
```

If control has reached the end of this code, and we are debugging without any saved state information, we can analyze the code to determine something about the value of the variable k before the **if** statement. For instance, if the current value of u is negative, we can deduce that the variable k should be less than zero at the beginning of this code, although we cannot say anything about its exact value. Taken far enough back, we might be able to determine ranges and constraints on sets of important variables, thus providing the user enough information to try forward and backwards execution to attempt to pinpoint the faulty code. This would be an especially valuable aid when debugging a program that has failed while running without the debugger tracing turned on. Techniques used in the development of the Mothra automatic test-case generator [DeMi87a, DeMi88a] appear adaptable to this problem.

Another promising approach involves examining test cases that execute correctly on the code under examination, and then constraining or parameterizing the values of variables at critical points in the code. By closely coupling the behavior observed under testing with our debugger, it may be possible to do backwards execution without any check-pointing at all. For instance, the current Mothra testing environment does extensive testing of control flow, domain

analysis, and predicate analysis for mutation testing [DeMi87b, DeMi86]. By analyzing test cases that perform correctly, and by analyzing the behavior of certain classes of mutations on those test cases, we may be able to deduce information about ranges of values and branches taken, without any saved state information [DeMi88b].

7. Conclusions

In this paper we have argued that an execution backtracking facility in interactive debuggers would be of significant help to users. With this, they would be able to match the debugging mechanism with their thought process of working backwards from the location where an error is manifested, and determining conditions under which the error could occur. We have also outlined a space-efficient approach to implementing execution backtracking, and compared it to approaches used by other systems that provide similar facilities.

One main advantage of our approach is that the space required can be bounded at compile time. This is in contrast to approaches used in other systems where large amounts of space may be required to debug long-running programs, and where space requirements, in general, cannot be bounded in advance.

We are in the process of developing a prototype debugger with backtracking capabilities as described in this paper. That debugger is being integrated with the Mothra software testing environment and will be used to further develop our ideas. We will also do further exploration about integrating backwards execution with testing in our prototype.

Acknowledgments

We would like to thank Richard A. DeMillo for discussions that helped clarify and motivate our research in this area. We are grateful for comments by Narain Gehani that aided us in formalizing our complexity bounds. The work in this paper was funded by a grant from the Purdue University/University of Florida Software Engineering Research Center (SERC).

References

Adam86.

Adams, Evan and Steven S. Muchnick, "Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations," *SOFTWARE-PRACTICE AND EXPERIENCE*, vol. 16, no. 7, pp. 653-669, July 1986.

Agra88a.

Agrawal, Hiralal and Eugene H. Spafford, "An Annotated Bibliography on Debugging," TECHNICAL REPORT SERC-TR-24-P, Software Engineering Research Center, Dept. of Computer Sciences, Purdue University, West Lafayette, IN, August 1988.

Agra88b.

Agrawal, Hiralal and Eugene H. Spafford, "An Execution Backtracking Approach to Program Debugging," TECHNICAL REPORT SERC-TR-22-P, Software Engineering Research Center, Dept. of Computer Sciences, Purdue University, West Lafayette, IN, June 1988.

Arch84.

Archer, James E. Jr., Richard Conway, and Fred B. Schneider, "User Recovery and Reversal in Interactive Systems," *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, vol. 6, no. 1, pp. 1-19, Jan. 1984.

Balz69.
Balzer, R. M., "EXDAMS: EXtendible Debugging and Monitoring System," *AFIPS PROCEEDINGS, SPRING JOINT COMPUTER CONFERENCE*, vol. 34, pp. 567-580, AFIPS Press, Montvale, N.J., 1969.

Bran83.
Branscomb, L. M. and J. C. Thomas, "Ease of Use: A System Design Challenge," in *Information Processing 83, Proceedings IFIP 9th World Computing Conference*, ed. R. E. A. Mason, pp. 431-438, Elsevier North-Holland, New York, 1983.

Carg86.
Cargill, Thomas A., "The Feel of Pi," *PROCEEDINGS OF THE WINTER USENIX CONFERENCE, DENVER, CO*, pp. 62-71, Jan. 1986.

DeMi86.
DeMillo, R. A. and E. H. Spafford, "The Mothra Software Testing Environment," *PROCEEDINGS OF THE 11TH SOFTWARE ENGINEERING WORKSHOP*, NASA Goddard Space Flight Center, Dec. 1986.

DeMi87a.
DeMillo, R. A. and A. J. Offutt, "Constraint Based Automatic Test Data Generation," TECHNICAL REPORT SERC-TR-5-P, Software Engineering Research Center, Dept. of Computer Sciences, Purdue University, West Lafayette, IN, Aug. 1987.

DeMi87b.
DeMillo, R. A., D. S. Guindi, K. N. King, and W. M. McCracken, "An Overview of the Mothra Software Testing Environment," TECHNICAL REPORT SERC-TR-3-P, Software Engineering Research Center, Dept. of Computer Sciences, Purdue University, West Lafayette, IN, Aug. 1987.

DeMi88a.
DeMillo, Richard A. and A. Jefferson Offutt, "Experimental Results of Automatically Generated Adequate Test Sets," *PROCEEDINGS OF 6TH PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE*, Portland, OR, Sept. 1988.

DeMi88b.
DeMillo, R. A., "Relating Testing and Debugging," TECHNICAL REPORT (TO BE RELEASED), Software Engineering Research Center, Dept. of Computer Sciences, Purdue University, West Lafayette, IN, August 1988.

Dunl86.
Dunlap, Kevin J., "Debugging with Dbx," *UNIX PROGRAMMERS MANUAL, SUPPLEMENTARY DOCUMENTS 1*, Computer Science Division, University of California, Berkeley, California, Apr. 1986. 4.3 Berkeley Software Distribution.

Goul75.
Gould, J. D., "Some Psychological Evidence on How People Debug Computer Programs," *INTERNATIONAL JOURNAL OF MAN-MACHINE STUDIES*, vol. 7, no. 1, pp. 151-182, Jan. 1975.

Haer83.
Haerder, T. and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *COMPUTING SURVEYS*, vol. 15, no. 4, pp. 287-317, Dec. 1983.

Hans71.
Hansen, W. J., "User Engineering Principles for Interactive Systems," *AFIPS PROCEEDINGS, FALL JOINT COMPUTER CONFERENCE*, vol. 39, pp. 523-532, AFIPS Press, Montvale, N.J., 1971.

Kats79.
Katsoff, H., "SDB: A Symbolic Debugger," *UNIX PROGRAMMER'S MANUAL*, AT&T Bell Laboratories, Murray Hill, N. J., 1979.

Luke80.
Lukey, F. J., "Understanding and Debugging Programs," *INTERNATIONAL JOURNAL OF MAN-MACHINE STUDIES*, vol. 12, no. 2, pp. 189-202, Feb. 1980.

Mara79.
Maranzano, J. and S. Bourne, "A Tutorial Introduction to ADB," *UNIX PROGRAMMERS MANUAL*, AT&T Bell Laboratories, Murray Hill, N. J., 1979.

McDo87.
McDowell, Charles E., David P. Helmbold, and Anil K. Sahai, "A Survey of Debugging Tools for Concurrent Programs," TECHNICAL REPORT UCSC-CRL-87-22, Baskin Center for Computer Engineering & Information Sciences, University of California, Santa Cruz, CA, Dec. 1987.

Rand75.
Randell, Brian, "System Structure for Software Fault Tolerance," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. SE-1, no. 2, pp. 220-232, June 1975.

Robb83.
Robbins, Arnold D., "The Design of a Passive Monitor for Distributed Programs," TECHNICAL REPORT GIT-ICS-83/21, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, Aug. 1983.

Schi81.
Schiffenbauer, R. D., "Interactive Debugging in a Distributed Computational Environment," MASTER'S THESIS, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, Aug. 1981.

Schw71.
Schwartz, Jacob T., "An Overview of Bugs," in *Debugging Techniques in Large Systems*, ed. Randall Rustin, pp. 1-16, Prentice-Hall, Engelwood Cliffs, New Jersey, 1971.

Teit72.
Teitelman, Warren, "Automated Programming: The Programmer's Assistant," *AFIPS PROCEEDINGS, FALL JOINT COMPUTER CONFERENCE*, vol. 41, pp. 917-921, AFIPS Press, Montvale, N.J., 1972.

Teit78.
Teitelman, Warren, *Interlisp Reference Manual, Fourth Edition,* Xerox Palo Alto Research Center, Palo Alto, CA, 1978.

Verh78.
Verhofstad, Joost S. M., "Recovery Techniques for Database Systems," *COMPUTING SURVEYS*, vol. 10, no. 2, pp. 167-195, June 1978.

Zelk71.
Zelkowitz, M. V., "Reversible Execution as a Diagnostic Tool," PH. D. DISSERTATION,

Dept. of Computer Science, Cornell University, Jan. 1971.

Zelk73.
Zelkowitz, M. V., "Reversible Execution," *COMMUNICATIONS OF THE ACM*, vol. 16, no. 9, p. 566, Sept. 1973.

A DEBUGGING ASSISTANT FOR DISTRIBUTED SYSTEMS

Daniel Hernández
Laveen Kanal
James M. Purtilo

ABSTRACT: We incorporate a fault localization capability into POLYLITH, a system that supports the interconnection of a distributed collection of heterogeneous software modules [Purt86]. To this end we apply techniques developed in the context of diagnosis of general technical systems. These techniques are based on explicit descriptions of the structure and behavior of the system to be diagnosed. The POLYLITH Module Interconnection Language (MIL) originally provides a description of software interconnectivity (structure), which is enhanced here by attributes specifying the high level behavior of the modules. Furthermore, the run-time support by the POLYLITH software bus gives us access to the actual behavior of the system under consideration. We are then able to determine a module or set of modules that must be faulty in order to explain the given observations. This system, called MOFALO, is implemented in Franz Lisp on a Sun Workstation. It consists of a modified syntax for the POLYLITH MIL, an "algebra" for describing the intended behavior within the example domains, and an implementation of deKleer and Williams' core fault localization algorithm [deWi87].

Daniel Hernández, a native of Colombia, is a 1986 graduate Friedrich-Alexander-Universität Erlangen in the Federal Republic of Germany. In 1987 he received his Master of Science degree in Computer Science from the University of Maryland at College Park, and is currently working on his doctorate at the Technische Universität in Munich, FRG.

Laveen Kanal is a Professor of Computer Science at the University of Maryland, where he is currently directing a research laboratory for machine intelligence and pattern analysis. In addition, he is Managing Director of L.N.K. Corporation, a firm doing management consulting, research and development, on problems in computer cartography, image processing, optical character recognition, automated inspection, and technology assessment. Dr. Kanal is an Associate Editor of *Pattern Recognition*, *Journal of Combinatorics, Information and System Science*, and *Pattern Recognition Letters*.

James M. Purtilo received his doctorate in Computer Science in 1986 from the University of Illinois at Urbana-Champaign, where he developed the "tool bus" paradigm for integration of heterogeneous software modules. Since then he has been on faculty in the Computer Science Department at the University of Maryland at College Park, where he continues to work on verification techniques based on software interconnection systems. In 1987 he was appointed to the University of Maryland's Institute for Advanced Computer Studies to study multiparadigm computing systems.

All correspondence goes to:
Dr. James M. Purtilo
Computer Science Department
University of Maryland
College Park, MD 20742

Office phone: (301) 454 1832
INTERNET: purtilo@mimsy.umd.edu

A DEBUGGING ASSISTANT FOR DISTRIBUTED SYSTEMS

Daniel Hernández *
Laveen Kanal †
James M. Purtilo ‡

We incorporate a fault localization capability into POLYLITH, a system that supports the interconnection of a distributed collection of heterogeneous software modules [Purt86]. To this end we apply techniques developed in the context of diagnosis of general technical systems. These techniques are based on explicit descriptions of the structure and behavior of the system to be diagnosed. The POLYLITH Module Interconnection Language (MIL) originally provides a description of software interconnectivity (structure), which is enhanced here by attributes specifying the high level behavior of the modules. Furthermore, the run-time support by the POLYLITH software bus gives us access to the actual behavior of the system under consideration. We are then able to determine a module or set of modules that must be faulty in order to explain the given observations. This system, called MOFALO, is implemented in Franz Lisp on a Sun Workstation. It consists of a modified syntax for the POLYLITH MIL, an "algebra" for describing the intended behavior within the example domains, and an implementation of deKleer and Williams' core fault localization algorithm [deWi87].

1 Introduction

Module interconnection languages have been used to provide a hierarchical description of application systems' designs for over a decade. The latest generation of MIL systems, which we refer to as being "polylithic," also employ a run time resource called a "tool bus" to assist in communication between distributed modules, coercion of data and dynamic reconfiguration of the design. A tool bus is an active agent, and already contains the application's structural specification in order to function. This in turn suggests that the tool bus is also a natural place to transparently instrument the application for purposes of debugging and maintenance.

This paper describes our investigation into debugging distributed systems at the module level by using an enhanced tool bus in the POLYLITH software interconnection system [Purt86]. Our debugging assistant, called MOFALO, utilizes the instrumentation features provided through the POLYLITH bus together with domain-specific specifications of desired application behavior. Based on these resources, the MOFALO assistant interacts with the user in order to precisely localize a module fault. The method we employ was originally developed in the context of diagnosing "technical systems," by which we mean any type of designed machinery such as electromechanical equipment, electronic circuits, or processing plants [deWi87].

---

* Inst. f. Informatik, Technische Universitaet Muenchen. Formerly at MIPAL, research supported by NSF grant ECS-8300799.

† Computer Science Department and Machine Intelligence and Pattern Analysis Lab (MIPAL), University of Maryland. Research supported by NSF grants ECS-8300799 and DCR-8504011.

‡ Computer Science Department and Institute for Advanced Computer Studies, University of Maryland. Research supported by Office of Naval Research contract N00014-87-K-0307.

The type of faults we seek to detect are not those that arise due to errors in a communication medium nor from node failure. Those are detected and handled in the tool bus itself. Instead, we assist developers in debugging the behavior of their modules' implementations. The scope of this fault localization assistance is limited to the module level, and does not replace but rather complements other debugging approaches that can be applied once a module or a set of modules has been identified as possibly faulty. Thus the debugging process can be sped up by avoiding detailed analysis of modules not likely to be responsible for a given fault.

The particular method we choose for suggesting potentially faulty modules is motivated by the similarity between software and technical systems. Both are human-made for a specific purpose and under given external constraints. Thus they share characteristics stemming from the human design process such as:

1. the hierarchical organization which typically results from problem decomposition;
2. the existence of detailed descriptions necessary to build the end product (in the case of software this may be the structural specification that is available in polylithic systems); and
3. a model of how the system is intended to behave (in the case of software this may be in the form of either requirements or test suites).

Specifically, our method is based on the POLYLITH interconnectivity description (enhanced by attributes specifying the high level behavior of the modules), together with the information provided by the run-time support of the POLYLITH software bus. Additional features of our approach allow us to use our high level behavior descriptions for "qualitative" simulation of otherwise unimplemented modules. In this way we allow early testing and rapid prototyping.

Section 2 reviews current approaches to software debugging, and defines the problem we solve in this project. Section 3 contains a capsule summary of the interconnection system on which this project was based, emphasizing the features of this system which are essential to support these debugging activities [PuRG87]. Section 4 describes our implemented debugging system in detail. This work is based on research initially performed in [Hern87]. Section 5 describes a sample of the notation used by designers to specify constraints concerning modular behavior.

2 Background

Approaches for debugging software depend on what knowledge is available for the system being analyzed, as well as on the experiences of those performing the analysis. When building machine assistants to partially automate the debugging process, we certainly have large amounts of knowledge to work with concerning the programming language, environment, and often the application domain as well. The key is to organize this information in such a way that the user's experiences with the system being diagnosed can be recognized and clearly presented in the context of known patterns of error [John86, Sevi87]. The sources of information to be organized, that is, the "input" to the debugging process, can vary widely: actual code or sets of input/output pairs can be examined, the structure of code or data can be abstracted and analyzed, and often both the requirements and formal specification of the program are available for use.

What is basic to any debugging scheme, however, is the need to embody some association between the appearance of a 'problem' and the location of some program construct which can ultimately be shown to manifest itself as that fault. Determining this association between symptom and cause is known as *fault localization.* Often the association is derived due to extensive reasoning about the representation of the program itself: A bottom-up or *code-driven* approach starts with

the code and builds, through pattern recognition or symbolic evaluation, an abstract representation thereof (e.g. PUDSY [Luke80], LAURA [AdLa80]). Alternately, a top-down or *problem-driven* approach begins with the known specification of the program guides an analysis through refinement of abstract structures (e.g. PROUST [John86]). The fundamental limitation of detailed analysis is its computational cost, since it approaches the cost of full verification as it becomes more sophisticated [Sevi87].

An alternative to this analysis of the program representations itself is to examine the manifestations of that program's execution, i.e. its "symptoms" as described in terms of discrepancies between expected and actual behavior. These types of fault localization vary from direct heuristic associations between symptoms and possible causes [Hara83], to controlled interactive backward traces [Shap83], to combination of diagnostic techniques with pattern recognition [Sedl83, Shap81], to message trace analysis [GuSe84]. Although full program analysis might create a very rich representation that will provide a better basis for debugging, it might also cost too much. On the other hand, the fault localization schemes based on program execution, while computationally more tractable, might only be able to coarsely localize a fault without being able to suggest a repair.

In general, both of the above approaches have merit, and neither yet seems to have outdistanced the other in the literature. However, our interest here is in debugging *distributed programs*. Here the task of debugging is usually complicated by a number of very pragmatic considerations, such as having multiple address spaces, more complex channels for notification of exceptions, and an additional dimension for problems, that of synchronization. In fact, depending on the execution environment, a buggy, distributed program can often have some subset of its component modules continue execution —showing signs of liveness to the user— when in fact some of its components have halted due to some hidden, fatal error.

The *diagnostic process* that we accept is described in very general terms as follows: A set of symptoms characterizes (not uniquely in general) a specific fault; conversely a fault labels its characteristic set of symptoms. A *fault* is thought to be an independent entity that influences particular components of the device. Those components impaired in their functionality are the ones that become manifest indirectly in the form of observable *symptoms*. Because the final goal of diagnosis is to reestablish the system in a functioning state, it is embedded in the more general framework of the so-called *repair problem* (that is, "determining that a program is buggy, identifying the bug and repairing it" [Sevi87]). Diagnosis in the narrower sense (that is, identifying the bug) we will call *fault localization* and is what we are concerned with in this project.

Little previous work has been done specifically to address the problem of localizing program faults specifically in a distributed configuration. To guide our initial effort, we have elected to utilize the experiences of previous research in medical and technical diagnosis. The interactive system we will describe emphasizes the diagnostic process called *envisionment* at the modular level, that is, the ability to imagine dynamic processes. The main step will be to record differences between the observed behavior of the real system and the behavior of the model gained by qualitative simulation, allowing the system to infer from those *structural discrepancies* and, hopefully, explain the observed malfunction. Our approach will work at the modular level without knowledge of the internal structure of modules, so that the only possible repair prescription is to *exchange* the module(s) found to be faulty. Hence the fault and the component are identified in this approach.

3 Overview of the Polylith System

There are several available systems for writing distributed programs. Only one of the them, our own POLYLITH software interconnection system, provides both the protocols for interconnection

of individual components plus the availability at execution-time of a complete structural (design) specification which, we will show, is necessary for localization in distributed programs. It is this POLYLITH system which we employ for our experiments in fault localization; indeed, to some extent it is the availability of this type of facility which initially suggested to us that fault localization might be successfully performed in distributed systems.

POLYLITH is a system that supports the interconnection of heterogeneous software modules and any associated hierarchical design [Purt86]. This is done by providing:

- a *module interconnection language* (MIL) to describe modules (in terms of so-called *minimum specifications*), their interfaces and their interconnectivity; and
- a *software bus* to support the run-time communication between modules, based on the MIL description.

Abstractly, a POLYLITH bus exists as an identifiable target to which modules can easily and transparently interface. Transitively a system's modules may then interface to one another. The intended interconnection between modules is *specified* by programmers through the POLYLITH MIL, and is *managed* at run time by an instance of some POLYLITH bus. Different instances of busses exist to serve different run-time needs independent of an application's structure, such as instrumentation or debugging. However, while an application's structure is described in this POLYLITH language, each component module's implementation can be written in whatever application language is appropriate (e.g. Pascal, C, Prolog, or Lisp).

An application system is thought of in terms of independent modules that interact with each other through specified interconnections to form a coherent whole. A *module* embodies a specific functionality and defines a number of *interfaces* or *ports* through which it can communicate with other modules. Furthermore, a set of attributes (for instance "implementation" information or algebraic specifications) can be associated with each module. In the current POLYLITH implementation, interfaces correspond to entry points in the code that can be called externally. Accordingly, the *interface pattern* specifying the structure of the transaction through the interface corresponds to the pattern of arguments of the call and their types.

Using the POLYLITH MIL, the actual interconnectivity (topology) of the system may be specified by instantiating modules (from their generic definitions) and enumerating the bindings between interfaces with compatible interface patterns. The resulting system corresponds directly to a simple attributed graph, where each node represents a module and the directed arcs between them represent bindings between interfaces. This graph analogy will assist us later in representing the distributed program to users.

The important concept here is the separation of descriptive assertions (about the modules and their interconnectivity) from their implementation (in terms of specific programming languages and actual communication media). This frees the modular design from considerations of low-level, language specific details, and also frees the individual implementations from the necessity of knowing the inner workings of other modules or the interconnection media (i.e., they can handle all function calls as if they were in the same process space and written in the same programming language). This separation is possible due to the software bus, which as an active substrate parameterized by the cluster specification, manages all communication between modules at run-time using a POLYLITH standard data representation protocol. To achieve execution, there must be an instance of a POLYLITH software bus (often referred to as the *message handler*, or just "MH".) The MH will start up the individual processes (based on information organized in an `implementation` attribute) and then act as a message delivery system for them. Optionally the MH will keep a logfile of all transactions done through the bus.

To summarize, POLYLITH provides us with a technology for interconnecting modules and easily mapping them onto a variety of host configurations. The modules can be implemented in differing languages, with the hosts connected by arbitrary communication media, and the set of underlying architectures can be heterogeneous. The features which are necessary for our fault localization activity are:

- the protocols embodying the software bus (or toolbus) can force messages to be serialized at need, and can transparently log these transactions;
- all such logged data will be in a standard representation (no more coercion should be necessary for the analysis to proceed);
- attributes can be associated with arbitrary components in the structural representation (we will use these to associate specifications concerning the correctness of those components, for purposes of debugging);
- and synchronization can be controlled by appropriate tools which interface to the bus.

In short, we can organize in one place both the actual, run-time transactions within our distributed program, as well as any specifications which can characterize the correctness of that set of transactions. That is, we can have the basic components needed for fault localization.

4 Fault Localization

We may now discuss fault localization in terms of the POLYLITH environment. From the user's point of view a failure occurs each time the actual behavior of a program does not correspond to the expected behavior. This could be the case for reasons ranging from simple syntax errors detected by a parser to deeply rooted design errors, where the program runs 'just fine' but produces the wrong answers. Recall from our earlier discussion that in general a fault is considered to be an 'independent entity' affecting only particular components of the system (i.e. modules of the program). What the parser or the user will be detecting are faults, i.e., manifestations of an error (or a set of errors) and not the error itself. This distinction is important not only to classify the different kinds of faults we will encounter, but also to recognize the limitations of our approach.

We shall distinguish between "intra-modular" and "inter-modular" errors. *Intra-modular* errors are likely to be programming-language dependent, but can range from syntax errors (e.g., illegal expressions, missing delimiters) to semantic mistakes and flaws in design. We will not be concerned here with these errors, since our model simply lacks the information necessary to do so. Hence we will instead be looking for *inter-modular* errors, which have ample opportunity to arise in distributed programming: (1) error conditions detected by the operating system at run time; (2) network and hardware related faults; (3) "classical" synchronization problems such as *deadlock*, *starvation* and unexpected order of events; (4) wrong results, which are not errors that could be detected by the operating system or the software bus; and (5) flaws in the modular design. Conventional error-handling mechanisms will be able to handle most faults in categories 1 and 2. Faults in category 3 can be detected in the POLYLITH tool bus. The category 5 is intrinsically outside the scope of model based fault localization, unless, of course, multiple redundant descriptions are available.

The remaining error category 4 is dealt with directly by our approach. In the earlier overview of POLYLITH, we described separately what will be the two main sources of information in the debugging process, that is, the MIL descriptions of the program under consideration, and the active communication substrate, i.e. the software bus.

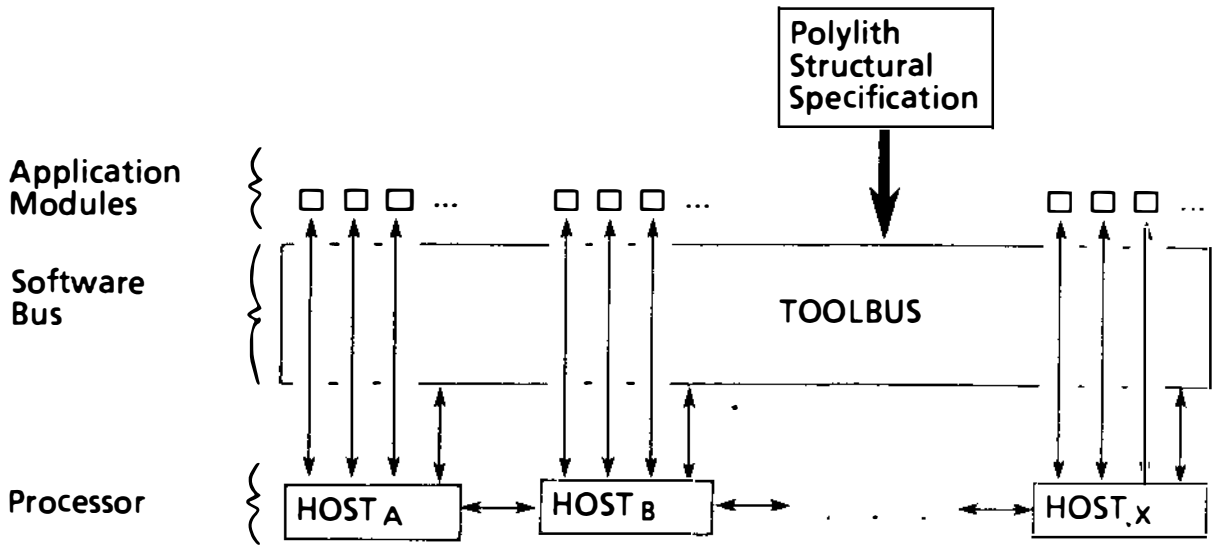


Figure 1: Organization of sample Polylith application program.

Figure 1 illustrates the organization of a sample POLYLITH program. The application's components are distributed for execution across a collection of host processors. Actual communication is performed by the available network interconnection hardware, but only as controlled by our toolbus and protocols. Therefore, we view the bus as being a logical layer between the application components and hosts (as drawn), even though the active entity itself is a separate process (or processes) executing on one (or some) of the host processors (much as a protocol nameserver).

Our assistant for fault localization, called MOFALO, is a program that is yet another logical layer on top of the existing POLYLITH bus. It has available to it the run-time IO of each individual module, the log of transactions sent among the individual components (as controlled by our toolbus protocol), and the structural specification describing the actual distributed program's topology. Further, using a specially altered POLYLITH toolbus, a user may control delivery of messages between distributed components (both stepping operations and ordering transactions). This organization is illustrated in Figure 2.

As is typical of testing systems, a mechanism for comparing the expectations (specification) and manifestations (run-time transactions) is required. MOFALO can function in an interactive mode, querying the user for each comparison. With the user as 'oracle' our system requires no additional domain information — all such domain information is encapsulated with the user. However, such interaction can become tedious, and users are notoriously good-natured oracles, being eager to agree when in doubt. Hence, the more useful situation in our system is when domain information can in fact be made available. In the case that assertions can be made available to describe the correctness of transactions, our system can evaluate each component based on the well-formedness of its input and output messages, and suggest where potential errors may lie. An important feature of MOFALO is that the order in which messages are checked is controlled by a localization algorithm intended to minimize the amount of searching necessary to localize the fault.

The MIL description must contain both structure and behavior information. In the original

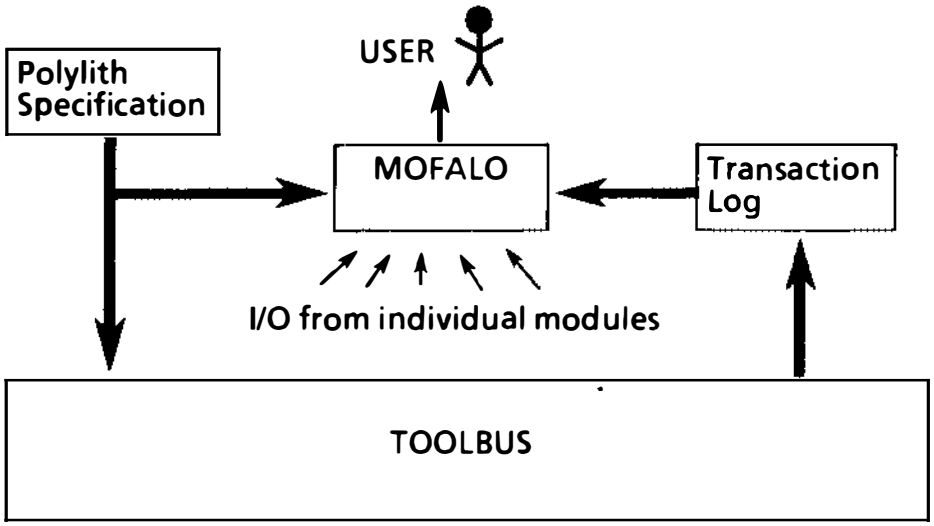


Figure 2: Sources of information for the MOFALO assistant.

POLYLITH system only the structural information was made available (although a mechanism for including additional, though uninterpreted, information was also available). In this project, we have extended the notation to allow for specification of behavior information as well, on a domain-by-domain basis. Hence, the MIL description plays two roles: it parametrizes the software bus at run-time and serves as the basis for qualitative simulation at fault localization time. Even in those cases where the behavior description is only partial or extremely simplified, our fault localization algorithm should work as long as it is possible to detect anomalies in what is transmitted between modules (i.e. "recognize conflicts") based on user evaluation.

The logfile actually provides more information than what we will need for our algorithm. It contains not only a protocol of the actual transactions, but also the order in which the modules's **read** and **write** requests to the software bus are done. Even though the order in which the individual services (processes) — once started by the message handler— are executed is arbitrary, the characteristics of the **read** and **write** requests (that is, blocking **read** operation with a non-blocking **write** operation with queuing if target not ready). guarantee that the order of transactions between any two given interfaces is preserved. However, in order to take advantage of this information for debugging purposes, we would also need to know the order in which the interfaces are used within a module. This information is not contained in the POLYLITH descriptions, but in the program code, which might not be available in a readable form (e.g., we may need to localize failures in a system components for which we do not have source code.) As suggested earlier, other debugging methods should be applied at the intra-modular level where this ordering information resides. At the level of abstraction of POLYLITH modules, the functionality of a module is likely to be a *relation* between inputs and outputs, where the precise order, in which the values needed to compute an unknown one are collected, is not relevant.

The fault localization algorithm that we have implemented in MOFALO was originally developed by deKleer and Williams [deWi87] for diagnosis in general technical (e.g. electro-mechanical)

systems. However, in contrast to their approach, which relies on a general "assumption-based truth maintenance system" (ATMS) to do most of the candidate generation, we have implemented the algorithmic part of the diagnostic mechanism directly and ignored fault probabilities as well as the optimization techniques they use. What follows is an overview of the algorithm, with full details of the implementation found in [Hern87].

The algorithm is based on a description of the system as a *network of objects*, in which values are *propagated* (during qualitative simulation) and *dependences* are recorded. In this object-oriented implementation, the *components* (nodes of the network — corresponding to our modules) as well as the *connectors* (edges of the network) are represented by objects, which are connected with each other by explicit pointers. In propagating values (realized by *message passing*), the visited components are assumed to function correctly. This fact is recorded in an *assumption set*, which is assigned to each value.

The components inherit a method for computing propagation values from the type of algebra used in the specification. This method operates on the algebraic specification of the module class to which a component belongs, as well as on the concrete inputs. This is similar to the "constraint propagation" paradigm, but here the components need not be constraints, i.e., relations (in the mathematical sense). Also, in contrast to the original constraint paradigm, values computed for a connector on different propagation paths are not only allowed, but form the starting point for fault localization. If two values of a connector do not agree, the union of their assumption sets results in a conflict set, which serves to identify possible candidates.

There is a distinction between *conflict* and *candidate* sets: The former is a set of components, of which at least one is malfunctioning. The latter is a set of components, whose simultaneous malfunctioning explains all symptoms. These definitions imply that each superset of a conflict or candidate is also a conflict or candidate, respectively. Therefore, it is enough to focus on minimal candidates (and conflicts), since they already characterize all possible candidates (conflicts). This leads to a variety of heuristics that constrain the combinatorial explosion of the propagation of multiple values.

The generation of minimal candidates from new, emerging conflicts and already known candidates is performed *incrementally* using an agenda and some queues. The minimality heuristics are built-in into the propagation mechanism (that is, into the methods of the generic components and connectors). A candidate *explains* a conflict if and only if both sets have a least one component in common. Whenever a new conflict is detected, the set of minimal candidates is modified in the following way: Each candidate that does not explain the new conflict is replaced by new candidates that result from supplementing it by each one of the conflict's components in turn.

The approach used to implement this algorithm is an incremental one that interleaves the conflict recognition and candidate generation phases. It is more efficient to generate new candidates sequentially based on new conflicts encountered and already known candidates.

The general outline of this variation of the algorithm is as follows:

- keep a record of the current set of minimal candidates (originally empty)
- whenever a new conflict is discovered:
    1. replace all minimal candidates that do not explain the new conflict by all their possible supersets containing exactly one more component
    2. eliminate all subsumed and repeated candidates
    3. check to see if the new candidates explain the conflict and repeat from 1 if they do not.

```
(algebra matcher
        (requires (input  = (?*x "-" ?*y "-" ?*z)))
        (ensures  (output = (?*x "-" ?*y ?*y "-" ?*z ?*z ?*z)))
)
```

Figure 3: Example of an assertion using our regular expression domain 'algebra.'

Since the propagation of values is done via message passing, the methods of the generic components and connectors must be extended to incorporate the minimality heuristics previously discussed.

The algorithm is implemented as a cyclic inspection of the global data structures followed by the corresponding processing of items found on those structures. Highest priority is given to the processing of new conflicts, since this leads to updates of the sets of minimal candidates and conflicts and is thus what prevents unnecessary propagations. If no new conflicts are available, values will be propagated by dequeuing the corresponding messages (tasks) from the agenda or external interactions will be processed (for example asking the user to perform measurement or consulting the logfile) as long as the interaction queue is not empty. If after all propagations are done, no unique candidate has been found, the system tries to find unmeasured connectors, whose inspection by the user or through consultation of the logfile might lead to discrimination among alternative candidates. If no more measurements are possible (i.e. if no more information can be gathered for a given structural description) the program terminates with a summary of results, either identifying a unique candidate or a set of possible suspects.

To summarize, the scope of our system is limited to fault localization and only in a category of failure where we identify faults with non-working modules. That is, no explanation as to what is the real cause of the bug will be given, nor will the assistant be able to correct it. Our approach is model-based working on a unique description (although alternative or even multiple algebraic descriptions are possible) and using only one core algorithm for fault localization. The reason for these limitations is that only limited information is available, or, viewed differently, that only with all the simplifying assumptions mentioned do we get a computationally tractable algorithm.

5 A Simple Example

To illustrate use of MOFALO we consider a simple application program which accepts a string, breaks it into substrings, applies transformations to each substring, then reassembles the resulting expressions into a data structure that is returned. While only a 'toy' problem as presented here, it is a (gross) simplification of the type of processing which is typical in many real applications, for instance, a modern editing system where transformations to an encapsulated parse tree might be applied in parallel.

Using POLYLITH it is trivial to map the structure of such an application onto a distributed host (or, for that matter, a tightly-coupled multi-computer.) The structural specification for this application, expressed using the POLYLITH MIL, is partially shown in Figure 4. However, since our interconnection system is based on an attributed graph model, it is usually more useful to consider a more graphic representation of this structure, such as given in Figure 5.

Our choice of domain algebra reflects another simplification made here in the interests of clarity. Since only strings are being passed as messages (in general, POLYLITH allows a richer collection of data types to be utilized), we have elected to use only a simple string matching language in order

describe the behavior of messages, or, more precisely, to describe what is required of an individual message to be considered well-formed. This domain 'algebra' gives us the ability to describe string transactions in terms of regular expressions. While not exercising an interesting and rich inference system, this choice does serve to illustrate how complete semantic properties of a component need not be captured in order to obtain the benefits of our approach to debugging. One example assertion (for the main module in our example) using regular expressions is shown in Figure 3, where the input is described as being a string x, a hyphen, a string y, another hyphen, followed by a string z; a correct pattern on the output transaction is declared to be a single copy of the string x, hyphen, two copies of the string y, hyphen, and finally three copies of the string z. This assertion concerning well-formedness is associated with a particular binding using **algebra** pointers in the MIL specification.

A user can debug a program with MOFALO either interactively or in a "batch" mode. That is, the debugging process can proceed while the task actually executes, or the program can be run in its entirety (or until the user identifies that an undesirable output has been generated) and then the localization activity can be initiated based on the stored log of transactions. Our example will illustrate an interactive session, where the MOFALO process (a lisp session with hooks into the POLYLITH bus) is started up on a Sun Microsystems workstation. Here, all interaction with the user for debugging appears in one window, and all interaction with the application modules is done in another window. In the following discussion, the term "probes" (a concept borrowed from circuit diagnosis) is used to describe instances of a special kind of printing module that are attached to the connectors like any other module. They are thought of as being attached to all module bindings in order to 'sense' and print the values passing through them. Otherwise we would not see any of the internal activities going on during the qualitative simulation phase.

First, a form of the structural description is loaded, so MOFALO will know what application to initiate (user input is italicized):

```
=> (load "m-rflow.cl")
;; Loading file "m-rflow.cl"
t
```

Consulting the global variables we see that as a result five components and several connectors have been generated:

```
=> components
(triple double single collector distrib)
=> connectors
(con-00461 con-00460 con-00459 con-00458 con-00457
con-00456 con-00455 con-00454)
=> module-list
(demo)
```

At the top level, the module **demo** is the name of our application — executing this main routine will cause MOFALO to initiate the task through the POLYLITH bus. The demo task immediately needs input (the first string to parse and send to distributed transformers); in the appropriate window, we provide the observable input value (**"do-re-mi"**). Since MOFALO will proceed with execution by default, we will see the final output value **"do-re-mimimi"**. However, the string **"do-rere-mimimi"** was expected — presumably our second transformer should have doubled its input instead of echoing its argument. We may re-execute the distributed program with the instrumentation and localization algorithm enabled:

```
(mofalo (demo))
Are there any observable system outputs?
(:input "do-re-mi")
Probe: con-00454 = (d o - r e - m i)
```

```
(module demo
    (algebra matcher
             (requires (input = (?*x "-" ?*y "-" ?*z)))
             (ensures (output = (?*x "-" ?*y ?*y "-" ?*z ?*z ?*z))) )
    (module distrib
        (implementation (binary  "./disp")
                        (machine  "leviathan"))
        (algebra matcher
                 (requires (input = (?*x "-" ?*y "-" ?*z)))
                 (ensures (outone = (?*x))
                          (outtwo = (?*y))
                          (outthree = (?*z))))
        (sink input string)
        (source outone string)
        (source outtwo string)
        (source outthree string) )
    (module collector
        (implementation  binary  "./join")
        (algebra matcher
                 (requires (one = (?*x))
                           (two = (?*y))
                           (three = (?*z)))
                 (ensures (stuffit = (?*x "-" ?*y "-" ?*z))))
        (sink one string)
        (sink two string)
        (sink three string)
        (source stuffit string) )
    (module double
        (implementation (binary  "./two")
                        (machine  "brillig") )
        (algebra matcher
                 (requires (stuff = (?*x)))
                 (ensures (out = (?*x ?*x))))
        (sink stuff string) (source out string) )
      (bind (distrib outone) (single stuff))
      (bind (distrib outtwo) (double stuff))
      (bind (distrib outthree) (triple stuff))
      (bind (single out) (collector one))
      (bind (double out) (collector two))
      (bind (triple out) (collector three))
    )
```

Figure 4: Structural specification for sample problem, using modified Polylith MIL notation (some component declarations omitted to save space here).

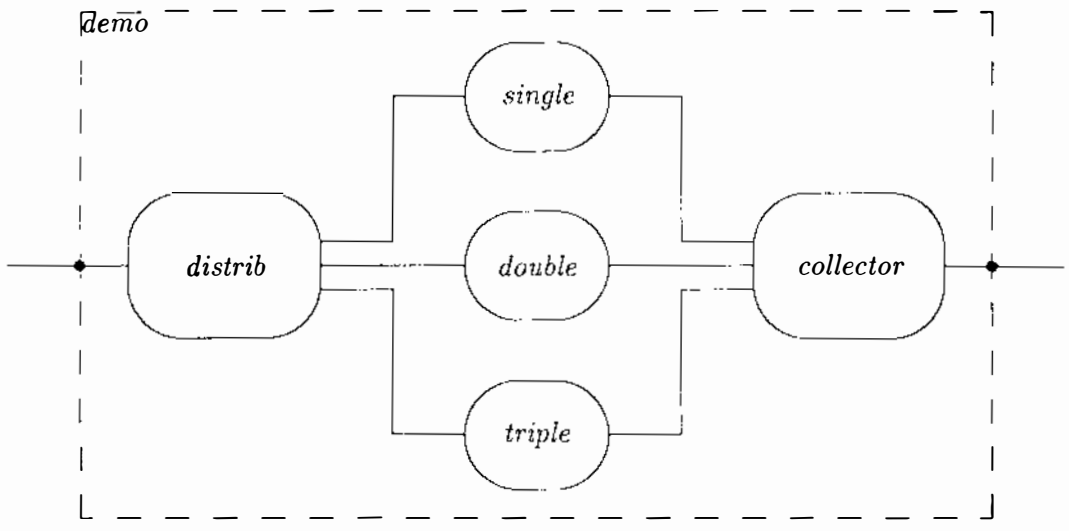


Figure 5: Graph representation for the sample problem.

```
Probe: con-00456 = (d o)

Propagating values...Probe: con-00457 = (r e)
Propagating values...Probe: con-00458 = (m i)
Propagating values...Probe: con-00459 = (d o)
Propagating values...Probe: con-00460 = (r e r e)
Propagating values...Probe: con-00461 = (m i m i m i)
Probe: con-00455 = (d o - r e r e - m i m i m i)
Propagating values...

:output ==>  "do-re-mimimi"

Probe: con-00455 = (d o - r e - m i m i m i)
Processing a new conflict...
((double) (triple) (collector) (distrib) (single))
((single distrib collector triple double))

Probe: con-00459 = (d o)

Propagating values...Probe: con-00460 = (r e)

(pattern= (?*x ?*x) datum= (r e))
```

After detecting a discrepancy between the simulated and actual behavior of the system, MOFALO lists the minimal candidates, followed by the minimal conflict set:

```
Propagating values...
Processing a new conflict...
((distrib) (collector) (double))
((double distrib collector))

Probe: con-00461 = (m i m i m i)
Propagating values...Probe: con-00456 = (d o)
Propagating values...Probe: con-00458 = (m i)
```

We have shown this with some of the monolog from the localization process included in the listing. Notice how the assistant has worked back through the log from the final (erroneous) output

to discover that the second transformer module has merely echoed its input, instead of echoing it twice.

This analysis by itself may not be conclusive in localization, however. The system orders the candidates according to their cardinality, assuming that candidates with less number of components are more likely, and then proposes additional tests to be performed on some modules to assist in the diagnosis. These measurements are taken on connectors that are chosen to yield the maximum information with respect to the set of candidates. Normally those connectors are the ones joining the candidates to each other. If however the first two candidates do not have connectors in common (as is the case here), the second and third (and so on) are considered. This, of course, is an ad hoc heuristic that might need to be revised in other applications. Measuring the value actually transmitted between **double** and **collector** eliminates the latter as a candidate. Similarly measuring the value transmitted between **distrib** and **double** determines **double** to be the culprit of the observed behavior. Once the set of minimal candidates contains a single element the system quits with that candidate as the suspect:

```
Choosing among (collector) (double)

Please enter value transmitted between:
(((double out) (collector two)))

"re"

Probe: con-00460 = (r e)

(pattern= (?*x ?*x) datum= (r e))

Processing a new conflict...
((double) (distrib))
((double distrib))

Propagating values...
Choosing among (double) (distrib)

Please enter value transmitted between:
(((distrib outtwo) (double stuff)))

"re"

Probe: con-00457 = (r e)
Probe: con-00460 = (r e r e)

Propagating values...
Processing a new conflict...
((double))
((collector double) (double))

MOFALO done
Suspect candidate is  (double)
```

Based on the specification for determining whether particular strings are well-formed, plus the execution-time log of transactions, MOFALO has determined that a fault in the module **double** can explain the error which was ultimately output. At no time was the actual implementation of a module ever inspected by the debugging system; presumably now that the error has been localized to a module, conventional intra-module debugging systems can be applied to that module on the appropriate processing element (which is known from the structural specification). Notice how the fact that each of these modules has been executing on a different machine has been hidden from the user.

6 Conclusion

We have described an approach to fault localization in distributed systems which is made possible by the power of the POLYLITH interconnection system. However, we feel the ideas presented are general, and that most available systems supporting interconnection of distributed components could take advantage of this work, albeit by incorporating a structural specification and nameserver capability similar to that of POLYLITH.

This initial effort has closely followed the approach used in diagnosis of general technical systems. One important consideration in these approaches is the cost of performing the analysis, and discussion concerning the complexity of our localization algorithm appears in [Hern87, HeKP88]. A more important consideration is whether the analogy between distributed programming and general electro-mechanical systems is strong enough that we would expect to find success by using similar approaches. One might argue that there is a fundamental difference between diagnosis of technical systems and software debugging: While diagnosis is performed to determine the cause of malfunction of an originally working artifact (i.e. with a sound/verified design), debugging is often viewed as being part of the design process itself. However, if one is willing to view the specification of a program (assumed to be bug free) as both the model and a perfect instance thereof (i.e. "the originally working artifact") then this difference appears to be relative and similar techniques can be used in both cases.

Once a module-level fault has been localized, traditional intra-module debugging techniques should be employed for diagnosing and repairing the component; part of our on-going research is to incorporate such capabilities in the existing MOFALO system. This requirement again points out one of the intrinsic differences of fault localization in software systems as opposed to the general technical systems from which much of our technique was suggested: Labeling a module or set of modules as faulty is not enough in the software context, since we do not have 'spare' modules at hand like in technical systems, where we could fix the problem by replacing the defective part. In software we must actually repair the defective module. Thus we must ultimately rely on other language-dependent debugging approaches to find the 'real' bug inside a module.

In a distributed environment there is often a problem which does not arise as frequently in single process-space programs, and this is that the methods used to instrument interaction between components often perturb the execution environment significantly. It is easier to find programs which function correctly when the instrumentation is enabled but which fail when the instrumentation is removed to improve performance. This situation does not occur within our system. Because of the way our POLYLITH interconnection system handles communication, exactly the same code and execution paths within an application component are used whether or not the instrumentation is enabled. This is not to imply that transaction logging is always done, restricting performance, but instead in our scheme logging is done by the bus, not the module. Therefore, use of our debugging system is transparent to the application code up to the point of timing. In fact, on POLYLITH busses designed for interconnection of modules on tightly-coupled multi-processors, the instrumentation can be performed transparently with respect to timing and message ordering, since the transactions can be captured and logged by a separate processing element with direct (shared) access to the relevant data structures in the run-time message handler.

The primary limitation of model-based fault localization lies in the models it uses. Only if the behavior specifications are rich enough will we be able to find conflicts and thus generate candidates. The problem of devising adequate description languages is related to efforts in many other areas of software engineering activity including program verification and very high level languages. However, our qualitative approach can proceed based on greatly simplified descriptions, whereas verification or automatic code generation require much more detailed specifications.

ACKNOWLEDGEMENT:

The authors wish to thank Mike Mazurek, Kathleen Romanik and Jack Callahan, all Graduate Research Assistants in the Computer Science Department at the University of Maryland, and also the NSQC referees for their many helpful comments and suggestions.

REFERENCES:

[deWi87] deKleer, J. and Williams, B.C. Diagnosing multiple faults. *AI Journal*, 32 (1987) 97-130.

[AdLa80] Adam, A., J.P. Laurent. Laura, a system to debug student programs. *AI Journal*, 15, (1980), 75-122.

[GuSe84] Gupta, N.K., and R.E. Seviora. An expert system approach to real-time system debugging. *Proceedings 1st Conf. on Artificial Intelligence Applications*, CS Press, Los Alamitos, (1984), 336-343.

[Hara83] Harandi, M.T. Knowledge-based program debugging: A heuristic model. *Proceedings of Softfair*, (1983), 282-288.

[HeKP88] Hernández, D., L. Kanal and J. Purtilo. Module fault localization in a software toolbus-based system. University of Maryland, Computer Science Department TR-2012, (April 1988).

[Hern87] Hernandez, D. Module fault localization in a software toolbus based system. *Master's Thesis.* University of Maryland, Department of Computer Science, (December 1987).

[John86] Johnson, W.L. Intention-based diagnosis of errors in novice programs. Morgan Kaufmann, Palo Alto, 1986.

[Luke80] Lukey, F.J. Understanding and debugging programs. *Intl. Journal of Man-Machine Studies*, (February 1980), 189-202.

[Purt86] Purtilo, J.M. A software interconnection technology to support specification of computational environments. Technical report UIUCDCS-R-86-1269, University of Illinois at Urbana-Champaign, 1986.

[PuRG87] Purtilo, J., D. Reed and D. Grunwald. Environments for prototyping parallel algorithms. *Proceedings of the Ninth International Conference on Parallel Processing* (August 1987), 431-438.

[Rich81] Rich, C. A formal representation for plans in the programmer's apprentice. *Proc. 7th. IJCAI*, (19810, 1044-1052.

[Sedl83] Sedlmeyer, R.L., et alia. Knowledge-based fault localization in debugging. *Proceedings ACM Software Eng. Symp. on High-Level Debugging*, (1983), 25-31.

[Sevi87] Seviora, R.E. Knowledge-based program debugging systems. *IEEE Software*, 4, 3,(1987), 20-32.

[Shap81] Shapiro, D.G. Sniffer: a system that understands bugs. *Technical Report MIT-AI Lab 638*, (June 1981).

[Shap83] Shapiro, E.Y. *Algorithmic program debugging.* MIT Press, Cambridge, MA (1983).

[Wate85] Waters, R.C. The programmer's apprentice: a session with KBEmacs. *IEEE Trans. on Software Eng.*, 11, 11, (1985), 1296-1320.

Specification

"Specifying Recoverable Objects"
Jeannette M. Wing, Carnegie Mellon University

"Requirements Specification Understanding and Misinterpretation"
Frank A. Cioch, Oakland University

"Specifications and Programs: Prospects for Automated Consistency Checking"
Albert L. Baker and John T. Rose, Iowa State University

Specifying Recoverable Objects

Jeannette M. Wing
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes the results of an exercise in writing formal specifications. The specifications capture the system-critical *recoverability* property of data objects that are accessed by fault-tolerant distributed programs. Recoverability is a "non-functional" property requiring that an object's state survives hardware failures.

This exercise supports the claim that applying a rigorous specification method can greatly enhance one's understanding of software's complex behavior. The specifications enabled us to articulate precisely questions about an unstated assumption in the underlying operating system, incompleteness in the implementation of recoverable objects, implementation bias in the language design, and even incompleteness in the specifications themselves.

Biographical Sketch

Jeannette M. Wing is an Assistant Professor of Computer Science at Carnegie Mellon University. She received her S.B. and S.M. in Electrical Engineering and Computer Science in 1979, and her Ph.D. degree in Computer Science in 1983, all from the Massachusetts Institute of Technology. Her research interests include formal specifications, language design, concurrent and distributed systems, and visual languages. She was a key designer of the Larch family of specification languages and is internationally recognized in the field of formal specifications. Her current research activity includes the design and implementation of Avalon/C++, a programming language for fault-tolerant distributed computing.

Specifying Recoverable Objects

Jeannette M. Wing[1]

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

1. Introduction

Formal specification languages have matured to the point where industry is receptive to using them and researchers are building tools to support their use. People use these languages for specifying the input-output behavior, i.e., *functionality*, of programs, but have largely ignored specifying a program's "non-functional" properties. For example, the functionality of a program that sorts an array of integers might be informally specified as follows: given an input array A of integers, an array B of integers is returned such that B's integers are the same as A's, and B's are arranged in ascending order. Nothing is said about the performance of the program like whether the algorithm for sorting should be $O(n)$ or $O(n^2)$. Performance is one example of a non-functional property.

In this paper, I will demonstrate the applicability of formal specifications to the non-functional property, *recoverability*. Recoverability requires that an object's state survives hardware failures. The correct behavior of these objects is fundamental to the correctness of the programs that create, access, and modify them. Sections 1.1 and 1.2 describe in more detail the context in which recoverable objects are used: fault-tolerant distributed systems. Section 2 describes how they are implemented at both the operating-system and programming-language levels.

The work described here is both theoretical and experimental in nature since the application of a formal (theoretical) specification language can itself be viewed as an experiment. Section 3 describes this specification exercise. The results of writing out specifications formally, summarized in Section 4, are extremely gratifying: they provide evidence that an existing specification language and method is suitable for describing a new class of objects; they validate the correctness of the design and implementation of a key part of an ongoing software development project; and not surprisingly, they demonstrate that the process of writing formal specifications greatly clarifies one's understanding of complex behavior.

1.1. Abstract Context: Fault-tolerant Distributed Systems

A distributed system runs on a set of nodes that communicate over a network. Since nodes may crash and communications may fail, such a system must tolerate faults; processing must continue despite failures. For example, an airline reservations system must continue servicing travel agents and their customers even if an airline's database is temporarily inaccessible; an automatic teller machine must continue dispensing cash even if the link between the ATM and the customer's bank account is down.

A widely-accepted technique for preserving data consistency and providing data availability in the

---

[1]This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864 (Amendment 20), under contract F33615-87-C-1499 monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB and in part by the National Science Foundation under grant CCR-8620027.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

presence of failures and concurrency is to organize computations as sequential processes called transactions. A *transaction* is a sequence of operations performed on data objects in the system. For example, a transaction that transfers \$25 from a savings account, *S*, to a checking account, *C*, might be performed as the following sequence of three operations on *S* and *C* (both initially containing \$100):

$$\{S = \$100 \wedge C = \$100\}$$

Read(*S*)
Debit(*S*, \$25)
Credit(*C*, \$25)

$$\{S = \$75 \wedge C = \$125\}$$

In contrast to standard sequential processes, transactions must be serializable, total, and recoverable. *Serializability* means that the effects of concurrent transactions must be the same as if the transactions executed in some serial order. In the above example, if two transactions, T1 and T2, were simultaneously transferring \$25 from S to C, the net effect to the accounts should be that S = \$50 and C = \$150 (that is, as if T1 occurred before T2 or vice versa). *Totality* means that a transaction either succeeds completely and *commits,* or *aborts* and has no effect. For example, if the transfer transaction aborts after the Debit but before the Credit, the savings account should each be reset to \$100 (its balance before the transfer began). *Recoverable* means that the effects of committed transactions survive failures. If the above transfer transaction commits, and a later transaction that modifies S or C aborts, it should be possible to ''roll back'' the state of the system to the previous committed state where S = \$75 and C = \$125.

It can be guaranteed that the integrity of the entire system is maintained if each object accessed within transactions is *atomic*. That is, each object is an instance of an abstract data type with the additional requirement that it must ensure the serializability, totality, and recoverability of all the transactions that use its operations. For example, as long as the bank account's Read, Debit, and Credit operations are implemented ''correctly,'' then any set of transactions that access the account will be serializable, total, and recoverable. The advantage of constructing a system by focusing on individual objects instead of on a set of concurrent transactions is modularity: one need only ensure that each object is atomic to ensure the more global atomicity property of the entire system.

Informally, a *recoverable* object is an object whose state can be restored to a previously ''checkpointed'' state if a node crash occurs. After a crash, a recoverable object is restored to a state that reflects only completed operations; the effects of operations in progress at the time of the crash are never observed. The restored state of a recoverable object must moreover reflect all operations performed by transactions that committed before the crash. Note that since a recoverable object's state may also reflect completed operations of aborted transactions, e.g., those active transactions that are automatically aborted at the time of the crash, recoverability is a weaker consistency property than totality. In the above bank account example, suppose immediately after the Debit a checkpoint of the system is made, and then while the Credit is in progress a crash occurs. The recoverable state of the system would be where S = \$75 and C = \$100.

The non-functional property of objects this paper focuses on is the recoverability aspect of atomic objects.

1.2. Concrete Context

The Avalon Project, co-managed by the author and Maurice Herlihy at Carnegie Mellon University, provides a concrete context for this work. We are implementing language extensions to C++ [10] to support application programming of fault-tolerant distributed systems. We rely on the Camelot System [9], also being developed at CMU, to handle operating-systems level details of transaction management, inter-node communication, commit protocols, and automatic crash recovery.

The formal specification language used in Section 3's specifications is Larch [7], though others such as Gypsy [5], VDM [2], Z [1], and OBJ [4], might also be suitable. The advantage gained in using Larch is the explicit separation between specifying state-dependent behavior (for example, side effects and resource allocation) and state-independent behavior (for example, the last-in-first-out property of stacks).

Where appropriate the details of Avalon, Camelot, and Larch are given, but the implications of the results of the specification exercise are independent of these projects.

A chart of the various people involved, the level (language or system) at which they are involved, and kinds of questions they address is shown below:

Person(s)	Language/System	Questions Addressed
Specifier	Larch	What is a recoverable object? What are the effects of its operations?
Language implementor	Avalon	How is a recoverable object represented in memory? How are its operations implemented?
Operating system builders	Camelot	How is memory managed? What protocol is used to recover memory after crashes?

2. Recoverable Objects

In order to appreciate the issues that arose in the specification of recoverable objects, in particular with respect to their implementation, it helps to understand their physical storage implementation (Camelot) and their programmer interface (Avalon).

2.1. The Operating System's View

Conceptually, there are two kinds of storage for objects: *local* storage whose contents are lost upon crashes, and *stable* storage whose contents survive crashes with high probability. (Stable storage may be implemented using redundant hardware [8] or replication [3].) Recoverable objects are allocated in local storage, but their state is written to stable storage so that recovery from crashes can be performed. If every recoverable object is *logged* to stable storage after modifying operations are performed on it in local storage, then its state may be recovered after a crash by "replaying" the log. Replaying the log is a sufficient method for restoring a object's state.

However, recovering the state of an object entirely from the log is a time-consuming operation. Camelot speeds up crash recovery by dividing local storage into two classes, volatile storage and non-volatile storage, and by distinguishing between two crash modes, node failures and media failures. In a *media failure*, both volatile and non-volatile storage are destroyed, while in a *node failure*, only volatile storage is lost. In practice, node failures are far more common than media failures. To optimize recovery from node failures, a protocol known as *write-ahead logging* [6] is used. An object is modified in the

following steps:

1. The page(s) containing the object are *pinned* in volatile storage; they cannot be returned to non-volatile storage until they are *unpinned*.
2. Modifications are made to the object in volatile memory.
3. The modifications are logged on stable storage.
4. The page(s) are unpinned.

The first step of the protocol ensures that the pages containing the object are not written to non-volatile storage while a modifying operation is in progress. This protocol ensures that a recoverable object can be restored to a consistent state quickly and efficiently. Upon crash recovery, the status of each transaction is determined, and by comparing what is in non-volatile storage to what is in stable storage, one can ''redo'' the effects of committed transactions and ''undo'' the effects of aborted ones. (For more details, see [9]). Notice that modifications must still be logged to stable storage to protect against the occurrence of a media failure.

2.2. The Programmer's View

The programmer's interface to a recoverable object is through the Avalon class header shown in Figure 2-1.

```
class recoverable {
public:

  void pin(int size);        // Pins object in physical memory.
  void unpin(int size);      // Unpins and logs object to stable storage.
}
```

Figure 2-1: Recoverable Class Definition

Informally, the *pin* operation causes the pages containing the object to be pinned, as required by the write-ahead logging protocol, while *unpin* logs the modifications to the object and unpins its pages. A recoverable object must be pinned before it is modified, and unpinned afterwards. For example if x is a recoverable object, a typical use of the pin and unpin operations within a transaction would be:

```
start {  // begin transaction
        ...
    x.pin();
        // modify x here
    x.unpin()
        ...
};      // end transaction
```

After a crash, a recoverable object will be restored to a previous state in which it was not pinned. Transactions can make nested *pin* calls; if so, then the changes made within inner *pin/unpin* pairs do not become permanent, i.e., written to stable storage, until the outermost *unpin* is executed.[2] Classes derived from *class recoverable* inherit *pin* and *unpin* operations, which can be used to ensure recoverability for

[2]Calls to *pin* and *unpin* must balance much like left and right parentheses should.

objects of the derived class.

The purpose of the specification exercise was to specify formally the effects of the *pin* and *unpin* operations, and hence the properties recoverable objects preserve.

3. Specifications

This section presents a sequence of three specifications as the different versions evolved during the specification process. The first version is more general than the second, but unimplementable. The third version removes an implementation bias that appears in both the first and second.

3.1. Version 1

The first version (see Figure 3-1) captures the following two properties of recoverable objects:

1. Each transaction can pin and unpin the same object multiple times.
2. Only at the last unpin does the object's value get written to stable storage.

First, we walk through the specification step-by-step. The following Larch *interface* specifications specify, using pre-and post-conditions, the effects of the *pin* and *unpin* operations:

```
pin = oper (x: recoverable)
  post x' = pn(x, self)

unpin = oper (x: recoverable)
  pre  pinned(x)
  post x' = un(x, self)
```

Self denotes the transaction (intuitively, the thread of control) that calls the operation. The precondition for *pin* is identically equal to true, meaning that a transaction can call *pin* in any state. The precondition for *unpin* requires that an object cannot be unpinned unless it is already pinned, given by the predicate of the same name.

The postconditions of the *pin* and *unpin* operations state what the changed value of a recoverable object is: x stands for the object's value in the initial state (upon invocation) and x' stands for its value in the final state (upon return). The postconditions make use of two auxiliary functions, specified in the Larch RecObj *trait*. *Pn* and *un* have the following signatures:

```
pn: R, Tid → R
un: R, Tid → R
```

where *R* and *Tid* are *sort* identifiers. *Pn* and *un* each take a recoverable object's value and a transaction identifier and return a (new) value for a recoverable object.

In any given state, a recoverable object's value is determined by the states of the transactions that have pinned it and the actual value of the object in memory. Thus, it is useful to "model" the value of a recoverable object as a pair of a Table and Memory.

```
Pair (R for T, Table for T1, Memory for T2)
```

where the **for** clauses rename sort identifiers (*T*, *T1*, and *T2*) that appear in one specification (*Pair*) used in another (*RecObj*).

The table component, indexed by transaction identifiers, keeps track of the number of times each transaction pins and unpins an object. The memory component keeps track of the actual value (with sort

```
class recoverable: interfaces
      based on R from RecObj

pin = oper (x: recoverable)
  post  x' = pn(x, self)

unpin = oper (x: recoverable)
  pre   pinned(x)      // cannot unpin something that's not already pinned
  post  x' = un(x, self)


RecObj: trait
  includes
      Pair (R for T, Table for T1, Memory for T2)
      TableSpec (Table for T, Tid for Index, Card for Val)
      Triple (Memory for T, M for T1, M for T2, M for T3, v for .first, n for .second, s for .third)
  introduces
      pn: R, Tid → R
      un: R, Tid → R
      pinned: R → Bool
  asserts for all (m: Memory, tb: Table, t: Tid, r: R)
      pn(<tb, m>, t) =
          if t ∈ tb                                  // already pinned?
          then <change(tb, t, eval(tb, t)+1), m>     // increment count
          else <add(tb, t, 1), m>                    // initialize it
      un(<tb, m>, t) =
          if eval(tb, t) = 1                         // if last unpin
          then <remove(tb, t), <m.v, m.v, m.v>>      // write to stable storage
          else <change(tb, t, eval(tb, t)-1), m>     // or just decrement count
      pinned(<tb, m>) = ~isEmpty(tb)
```

Figure 3-1: Specification of Class Recoverable: Version 1

M) of the object, as stored in each of the three levels of storage: volatile (*v*), non-volatile (*n*), and stable (*s*).

TableSpec (Table **for** T, Tid **for** Index, Card **for** Val)
Triple (Memory **for** T, M **for** T1, M **for** T2, M **for** T3, v **for** .first, n **for** .second, s **for** .third)

The meaning of *pn* is given by the following equation:

```
pn(<tb, m>, t) =
  if t ∈ tb
  then <change(tb, t, eval(tb, t)+1), m>
  else <add(tb, t, 1), m>
```

If the object (the pair *<tb, m>*) is already pinned by the given transaction (*t*), then *t*'s count is incremented in the table; otherwise a new entry is added to the table where the count is initialized to 1.

The meaning of *un* is as follows:

```
un(<tb, m>, t) =
  if eval(tb, t) = 1
  then <remove(tb, t), <m.v, m.v, m.v>>
  else <change(tb, t, eval(tb, t)-1), m>
```

Upon unpinning an object, for a given transaction (*t*), if its count of pins is down to 1, the object's value in volatile storage should be written to non-volatile and stable storage; otherwise, the count should merely be decremented by 1 and no change should be made to memory.

Putting all these pieces together results in the full specification shown in Figure 3-1. The Appendix contains the *Pair*, *TableSpec*, and *Triple* specifications.

3.2. Version 2

The specification of the previous section was shown to the implementor of *class recoverable* (Figure 2-1) in order to verify that indeed the implementation satisfies the specification. The implementor immediately noticed what he thought was an error in his implementation: The specification permits different transactions to pin the same object at the same time, whereas the implementation does not. The implementor proceeded to change his implementation to satisfy the specification, but then realized that the specified semantics was unimplementable! The underlying operating system (Camelot) forbids more than one transaction to pin an object (as represented as pages in volatile memory) at once. It assumes that any transaction pinning an object will modify that object and thus would want to prevent any other transaction from simultaneously accessing that object. A pinned object is a write-locked one as well. Thus, it was impossible to implement the less restrictive, but desired, semantics of *pin*; in short, the specification was "correct," but unimplementable.

The revised specification (Figure 3-2), which is more restrictive but implementable, captures this third property of recoverable objects:

3. Only one transaction can pin an object at once.

This specification is simpler to understand than the previous one because there is less information to keep track of. In essence, the table of transaction identifiers and their corresponding pin counts reduces to a single transaction and its count.

The specifications for *pin* and *unpin* change slightly:

```
class recoverable: interfaces
      based on R from RecObj

pin = oper (x: recoverable) signals (already_claimed)
  post  x' = pn(x, self) ∧
        x.trans ≠ self ⟹ signal already_claimed

unpin = oper (x: recoverable)
  pre   pinned(x) ∧ x.trans = self
     post  x' = un(x, self)


RecObj: trait
  includes
     Triple (R for T, Tid for T1, Memory for T2, Card for T3, trans for .first, count for .third)
     Triple (Memory for T, M for T1, M for T2, M for T3)
  introduces
     pn: R, Tid → R
     un: R, Tid → R
     pinned: R → Bool
  asserts for all (m: Memory, m1, m2, m3: M, c: Card, t1, t2: Tid)
     pn(<t1, m, c>, t2) =
        if c > 0                                  // is already pinned?
        then if t1 = t2                           // by same transaction
                then <t1, m, c+1>                 // increment count
                else <t1, m, c>                   // otherwise, leave unchanged
        else <t2, m, 1>                           // initialize it
     un(<t1, <m1, m2, m3>, c>, t2) =
        if t1 = t2                                // don't have to check if pinned already
        then if c = 1                             // if last unpin
                then <t1, <m1, m1, m1>, 0>        // write to stable storage
                else <t1, <m1, m2, m3>, c-1>      // or just decrement count
        else <t1, <m1, m2, m3>, c>                // no change
     pinned(r) = r.count > 0
```

Figure 3-2: Specification of Class Recoverable: Version 2

```
pin = oper (x: recoverable) signals (already_claimed)
  post  x' = pn(x. self) ∧
        x.trans ≠ self ⟹ signal already_claimed

unpin = oper (x: recoverable)
  pre  pinned(x) ∧ x.trans = self
  post x' = un(x, self)
```

Pin might terminate with an error condition signaled to the invoker to indicate that the object to be pinned is already pinned by some other transaction. *Unpin* requires not only that its argument is already pinned, but that it is pinned by the calling transaction.

Since concurrent pins by different transactions are not allowed, it is unnecessary to keep track of a table of pin counts per transaction. It suffices to associate with a recoverable object, a single transaction identifier, its value in memory, and a pin count:

```
Triple (R for T, Tid for T1, Memory for T2, Card for T3, trans for .first, count for .third)
```

Assume initially that each recoverable object, x, is unpinned, i.e., x.count = 0.

The auxiliary functions, *pn* and *un*, change accordingly:

```
pn(<t1, m, c>, t2) =
  if c > 0
  then if t1 = t2
         then <t1, m, c+1>
         else <t1, m, c>
  else <t2, m, 1>
```

If the count (c) is non-zero, then the object must be pinned. If the object is pinned by a transaction ($t1$) that is the same as the transaction ($t2$) attempting to pin the already pinned object, then the count is incremented; otherwise, the object is left unchanged. If the object is not already pinned, then its value is initialized with the pinning transaction's identifier and a count of 1.

```
un(<t1, <m1, m2, m3>, c>, t2) =
  if t1 = t2
  then if c = 1
         then <t1, <m1, m1, m1>, 0>
         else <t1, <m1, m2, m3>, c-1>
  else <t1, <m1, m2, m3>, c>
```

Unlike for *pn*, it is unnecessary for *un* to check if the object is already pinned since the precondition of *unpin* checks for this case. So *un* first checks to see if the transaction ($t1$) that currently has the object pinned is the same as the unpinning transaction ($t2$). If so, then if there is only one outstanding call to pin ($c = 1$), the value of the object in volatile storage is written to non-volatile and stable storage; otherwise, the count is decremented. If the unpinning transaction is different from the pinning one, then no change is made.

An Aside for Larch Readers

A typical use of *class recoverable* is to define a derived class for a recoverable type of object, say *class rec_foo*. If *foo* is the sort identifier associated with values of objects of type *rec_foo*, then the identifier M that appears in the *RecObj* specification would be renamed with *foo*. That is, the header for the Larch interface specification for a rec_foo class would look like:

```
class rec_foo: interfaces
    based on R from RecObj (foo for M)

    //... specifications of operations for rec_foo objects ...
```

3.3. Version 3

Notice that nowhere in the previous specification is the distinction between non-volatile and stable storage used. For example, when an object is finally unpinned, its value is written out to both non-volatile and stable storage:

```
un(<t1, <m1, m2, m3>, c>, t2) =
    ...
        then <t1, <m1, m1, m1>, 0>
    ...
```

The second two components of Memory are treated identically. In unpinning an object, it is necessary that stable storage be updated using volatile storage's value, but writing out to non-volatile storage is strictly not necessary.

This observation reveals an implementation bias in the specification. The underlying operating system implements memory as a three-level storage hierarchy, and uses the write-ahead logging protocol to exploit the distinction between volatile and non-volatile storage for crash recovery. Recall, however, that conceptually a recoverable object has only two possible ''values'': that in volatile storage and that in stable storage. It suffices to consider only a two-level storage hierarchy with just volatile and stable storage. The change to the previous specification is trivial since Memory simply becomes a pair:

```
Pair (Memory for T, M for T1, M for T2, v for .first, s for .second)
```

and *un* changes accordingly:

```
un(<t1, <m1, m2>, c>, t2) =
    ...
        then <t1, <m1, m1>, 0>
        else <t1, <m1, m2>, c-1>
    else <t1, <m1, m2>, c>
    ...
```

4. Observations

The different versions of the specification made it possible to articulate precisely questions about the semantics of recoverable objects as well as questions about the implementation. The feedback between the specifier and implementor and between the specifier and language designers helped everyone gain insight about the implementability of the desired semantics, incompleteness in the current implementation, implementation bias in the language design, and even incompleteness in the specifications as presented.

4.1. Unstated Assumption

The major observation as a result of this specification exercise is that the specification helped identify an unstated and critical assumption in the underlying operating system that was reflected in the implementation. The implementation precluded the possibility of concurrent pins by different transactions. The underlying system forbids this situation because it assumes that any transaction that pins an object intends to modify it.

This assumption reflects a key point at the operating-system level where recovery and synchronization of objects are inseparable. Without concurrency, one can give a meaning to recoverability; without recovery, one can give a meaning to the correct synchronization of processes. But to support both, there are points when one must consider both recovery and synchronization together. Here is exactly one of those points. Synchronization of concurrent, modifying transactions is built into the meaning of recoverability of objects. This point was not well understood by either the language designers or the language implementors because the assumption was never stated by the underlying operating system builders. Only through this specification exercise and subsequent discussion between the language implementors and system builders was this point clarified.

4.2. Incompleteness in the Implementation

When presented with the specification of the *unpin* operation (any version), the implementor was asked whether the precondition on *unpin* (requiring that the object be pinned and that the check is with respect to the calling transaction) could be removed. That is, should the responsibility of checking the stated precondition be on the caller of *unpin* or the implementor? Currently, the responsibility lies with the caller; however, it could easily be checked at runtime as part of the implementation. If the object is not pinned or pinned by some other transaction, an appropriate error message could be signaled to the caller, much like the error condition signaled in the *pin* opeation. The implementor was alerted to this assymmetry in handling error conditions only when the formal specification was presented to him.

4.3. Implementation Bias in the Language Design

The specification also revealed a subtle point of misunderstanding between the language designers and language implementor. *Class recoverable* is actually implemented to provide a stronger property, operation-consistency, than just recoverability. *Operation-consistency* requires that an object be restored to some consistent state that reflects all operations of committed transactions plus some prefix of the sequence of operations performed on the object by transactions active at the time of a crash. Since the implementation supported this stronger property and since the designers never carefully defined (that is, specified) recoverability, the meaning of recoverability was confused with the implementation of recoverability; thus, until this specification exercise was performed, the language designers believed that operation-consistency was inherent to recoverability.

The nondeterminism inherent in this stronger definition would force the specification to keep track of a set of possible values (each representing a prefix of operations of uncommitted transactions) in stable storage (the third component of Memory in Figure 3-2) rather than a single value. When the object's state is restored upon recovery, any one of the values in this set would correctly represent a previous operation-consistent state. One would additionally need to ensure that the restored states of all objects reflect the same prefix of operations of all uncommitted transactions. For example, if transaction T were active at the time of the crash and states of objects x and y are restored, if some prefix of T's operations is reflected by the x's restored state, then the same prefix must also be reflected in y's. Note that specifying this property cannot be done locally, i.e., per object; it is inherently a global property that involves the states of all objects in the system. One would specify a system-wide operation, *recover*, which would refer to the recovered, operation-consistent states of all the system's objects.

4.4. Incompleteness in the Specification

As is, the specification for *class recoverable* is not complete: initialization of a recoverable object is unspecified. Informally, a recoverable object is initially some block of memory with no associated transaction identifier (and of course no pin count) and no initial value. No transaction identifier is associated with a recoverable object until it is first pinned. Allocation of memory should be specified in the postcondition for a separate *create* operation:

```
create = oper () returns (x: recoverable)
  post new x
```

where "**new** x" is a special Larch assertion stating that *x* denotes some previously free block of memory. Also, either *pin*'s precondition should require that its argument has been previously allocated (making it the responsibility for the caller to check), or the auxiliary function *pn* should be modified accordingly (making it the responsibility of the implementor to check).

5. Concluding Remarks

In some sense the details of the problems discussed in the previous sections are less interesting than the insights gained from undertaking the process of rigorously specifying recoverability. This process enabled us to clarify fuzzy notions about recoverable objects; and to state precisely problems revealed in the specification, design, and implementation and to resolve their discrepancies.

Since this specification exercise was performed in the context of an ongoing large software development project, it was especially rewarding to identify points of confusion between desired and implementable semantics, to discover incompleteness in the implementation, and to separate out implementation biases from the design. A language like Avalon has more complex semantics than a standard sequential programming language; knowing early on that a fundamental part of its semantics is implemented correctly is a tremendous reassurance to us and future Avalon programmers. As language implementors, we promise to provide certain properties of the built-in classes like *class recoverable* so that when people use Avalon they need not worry that some error they find in their code might in fact be an error in ours. In particular, recoverability is a nontrivial, system-critical property of objects. The rest of the Avalon language class hierarchy derives from *class recoverable*, both in defining other built-in classes like *class atomic*, and in defining user-defined classes like recoverable strings or atomic queues. It is still impractical and unreasonable to specify formally large software systems completely, but the benefits of tackling smaller, system-critical pieces are large.

Finally, as mentioned in the introduction, we are able to demonstrate that formal specification techniques can be extended naturally to specify non-functional properties like recoverability. We intend to continue this specification exercise for the other built-in and user-defined classes of Avalon, in particular those that support other aspects of the *atomicity* property of objects.

Acknowledgments

I thank Maurice Herlihy for helping me better understand recoverability, David Detlefs for actually implementing recoverable objects, and Mathew Brozowski for his interest in specifying their behavior.

I. Other Specifications Used

```
TableSpec: trait
  introduces
      new: → Table
      add: Table, Index, Val → Table
      _∈ _: Index, Table → Bool
      remove: Table, Index → Table
      eval: Table, Index → Val
      change: Table, Index, Val → Table
      isEmpty: Table → Bool
  asserts
      Table generated by (new, add)
      Table partitioned by (_∈ _ , eval)
      for all (t: Table, ind, ind1, Index, val, val1: Val)
          ind ∈ new = false
          ind ∈ add(t, ind1, val) = (ind = ind1) | ind ∈ t
          eval(add(t, ind, val), ind1) = if ind = ind1 then eval else eval(t, ind1)
          remove(add(t, ind, val), ind1) = if ind = ind1
                                then t
                                else add(remove(t, ind1), ind, val)
          change(add(t, ind, val), ind1, val1) = if ind = ind1
                                then add(t, ind, val1)
                                else add(change(t, ind1, val1), ind, val)
          isEmpty(new) = true
          isEmpty(add(t, ind, val)) = false
  implies converts (_∈ _, remove, eval, change)
      exempting eval(new, ind), remove(new, ind), change(new, ind, val)

Triple: trait
  introduces
  <_, _, _>: T1, T2, T3 → T
  _.first: T → T1
  _.second: T → T2
  _.third: T → T3
  asserts
      T generated by (<_, _, _>)
      T partitioned by (first, .second, .third)
      for all (a: T1, b: T2, c: T3)
          <a, b, c>.first = a
          <a, b, c>.second = b
          <a, b, c>.third = c

Pair: trait
  introduces
  <_, _>: T1, T2 → T
  _.first: T → T1
  _.second: T → T2
  asserts
      T generated by (<_, _>)
      T partitioned by (.first, .second)
      for all (a: T1, b: T2)
          <a, b>.first = a
          <a, b>.second = b
```

References

[1] J.R. Abrial.
The Specification Language Z: Syntax and Semantics.
Technical Report, Programming Research Group, Oxford University, 1980.

[2] D. Bjorner and C.G. Jones (Eds.).
Lecture Notes in Computer Science. Volume 61: *The Vienna Development Method: the Meta-language.*
Springer-Verlag, Berlin-Heidelberg-New York, 1978.

[3] D. S. Daniels.
Distributed Logging for Transaction Processing.
In *Proceedings of the 1987 ACM Sigmod International Conference on Management of Data.* Association for Computing Machinery, San Francisco, CA, May, 1987.

[4] J.A. Goguen and J.J. Tardo.
An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications.
In *Proceedings of the Conference on Specifications of Reliable Software*, pages 170-189. Boston, MA, 1979.

[5] D.I. Good, R.M. Cohen, C.G. Hoch, L.W. Hunter, and D.F. Hare.
Report on the Language Gypsy, Version 2.0.
Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, The University of Texas at Austin, September, 1978.

[6] J. Gray.
Notes on Database Operationg Systems.
In *Lecture Notes in Computer Science.* Volume 60: *Operating Systems: an Advanced Course.* Springer-Verlag, Berlin, 1978.

[7] J.V. Guttag, J.J. Horning, and J.M. Wing.
The Larch Family of Specification Languages.
IEEE Software 2(5):24-36, September, 1985.

[8] B. Lampson.
Atomic transactions.
Lecture Notes in Computer Science 105. Distributed Systems: Architecture and Implementation.
Springer-Verlag, Berlin, 1981, pages 246-265.

[9] A. Spector, J. Bloch, D. Daniels, R. Draves, D. Duchamp, J. Eppinger, S. Menees, D. Thompson.
The Camelot Project.
Database Engineering 9(4), December, 1986.

[10] B. Stroustrup.
The C++ Programming Language.
Addison-Wesley, Reading, Massachusetts, 1986.

REQUIREMENTS SPECIFICATION UNDERSTANDING AND MISINTERPRETATION

Frank A. Cioch
Department of Computer Science and Engineering
Oakland University
Rochester, Michigan 48063

ABSTRACT

Because requirements specifications are used in the planning phase of the software life-cycle, misinterpretations of the specification are often as important as the understanding of it. In this paper we give the results of a study that was performed to assess both understanding and misinterpretation of requirements specifications for both programmers and nonprogrammers. Two languages were studied : Structured Analysis and structured English.

The results showed that for programmers, structured English resulted in significantly greater understanding and significantly fewer misinterpretations than did Structured Analysis. For nonprogrammers, there was not a significant difference in the understanding of the languages, but structured English resulted in significantly more misinterpretations than did Structured Analysis.

BIOGRAPHICAL SKETCH

Frank Cioch is an assistant professor of engineering at Oakland University, where he teaches courses in software quality and software engineering. His research interests include software engineering and human/computer interaction. In particular, he is interested in the impact of design methods on structural quality and evaluating and measuring software understandability, maintainability and reusability. Cioch received a Ph.D. in computer and communication sciences from the University of Michigan.

REQUIREMENTS SPECIFICATION UNDERSTANDING AND MISINTERPRETATION

Frank A. Cioch
Department of Computer Science and Engineering
Oakland University
Rochester, Michigan 48063

INTRODUCTION

Key components of requirements specification understandability are the concepts of readability and ambiguity and the associated roles played by formal and informal specifications [1, 2, 3]. An informal specification is thought to have high readability but also high ambiguity. The amount of material that can be easily understood is thought to be greater than can be achieved with a comparable formal specification. However, due to the high degree of ambiguity, the possibility of misinterpreting material is also thought to be greater. A formal specification is thought to reduce the possibility of misinterpreting material through the elimination of ambiguity, but understandability is also believed to be lower.

Requirements specification understanding vs. misinterpretation is particularly problematic because of the diverse backgrounds of the individuals who must read the specification [4, 5, 6]. Because the requirements specification document is used to ensure that the system being constructed is the one that the user wants, both designers and users must read the specification.

Ideally, the user will have high understanding of the specification with few misinterpretations. Intuitively, it would appear that a formal language would be ideal for the designer but an informal language would be ideal for the user. However, if the ambiguity associated with an informal language leads to misinterpretation of the specification, an informal specification might not be ideal for the user. A user with low understanding does not know what the proposed system will be like. This can be detected and hopefully corrected before design proceeds. However, misinterpretation of what the specification states results in a mismatch between the system that the user believes is going to be constructed and the system that actually will be constructed by the designer. In such a case, design proceeds and the problem is not detected until later in the life-cycle, when it is more expensive to resolve.

Despite the fact that these issues are important, they are difficult to address experimentally because of measurement difficulties. Incorrect responses in a performance test result from two sources: 1) the participant doesn't know the answer and is aware of it (indicating a lack of understanding), and 2) the participant gives the wrong answer but believes it is correct (indicating misinterpretation). It is difficult to obtain measures of both understanding and misinterpretation.

In this study, measures of both understanding and misinterpretation were developed and an experiment was performed to assess understanding and misinterpretation of requirements specifications.

EXPERIMENTAL METHOD

Experimental Design

Two languages were used in the experiment: Structured Analysis and structured English. Structured English is thought to be a good mixture of formalism and informalism because presumably it is interpreted as a formal computer program by programmers and as informal technical English by nonprogrammers [2].

Structured Analysis uses data flow diagrams rather than structured English to describe high level processes. Data flow diagrams, a technical graphical language, may also be a good mixture of formalism and informalism. Like a computer program, data flow diagrams are formal in that particular rules exist for reading and interpreting them. A key rule is that data flow diagrams show flow of data, not of control [6]. The workings of the system are presented as seen by the data, not the data processors. However, like structured English, data flow diagrams can also be read informally, without knowledge of the rules of interpretation. It has been suggested that not only can data flow diagrams be read by users, but they make it easy for the user to see if the system being constructed is the one that is desired [6].

Structured Analysis uses a combination of data flow diagrams for the description of high-level processes, structured English for the description of low-level processes, and a data dictionary for the description of the data interfaces of the processes in the data flow diagrams. In an effort to ensure that all differences in understanding could be attributed to the use of either structured English or data flow diagrams for the description of high-level processes, the specifications were similar in every respect except high-level process description. One such process written using both languages, Register-A-Student, is given in Appendix A. The specifications in each of the two languages had the same hierarchical functional decomposition, the same structured English process descriptions for the low-level processes, and the same data dictionaries.

Given the role of requirements specifications in the software life-cycle, the dimension of programming expertise was studied along with the two languages. Specifications in the two different languages were presented to both programmers and nonprogrammers. This resulted in a 2x2 factorial design with 20 participants in each of the 4 cells. Each participant saw one specification. None of the programmers who participated in the experiment were familiar with data flow diagrams and hence, with Structured Analysis.

Participants

The participants in the experiment were 80 undergraduate students at the University of Michigan. The 40 nonprogrammers, 16 males and 24 females, represented 20 different majors. Thirty-four of the 40 nonprogrammers were juniors or seniors. The 40 programmers in the experiment had taken an average of five courses which involved programming and knew an average of five programming languages, but were not familiar with Structured Analysis or data flow diagrams. All but two of the programmers, 30 males and 10 females, were juniors or seniors.

Experimental Materials

Each participant in the experiment was presented with a package containing four items: an Introduction, Training Material, a Specification, and a Test Instrument. The Introduction was written so that the participants would read the Specification from the same perspective as the software buyer. Participants were instructed to imagine that they were responsible for deciding

whether or not to contract for the development of a proposed student registration system. The Introduction concluded with a summary list of the functions that the system was expected to perform. The participants were told that it was their responsibility to ensure that the proposed system performed these functions.

Training material was written for each of the two languages, with as much similarity as possible between the training material for each language. The training material included a short system specification along with a narrative which walked the participant through the specification. In the process of walking the participant through the system, the narrative emphasized the kinds of things that a prospective buyer should look for.

The student registration system was chosen as the subject matter of the study because it was a subject with which the participants, university students, had familiarity. An informal survey conducted by the author showed that students have a great deal of knowledge about the problem domain, class registration, but little knowledge of the kinds of things that must be done by a computerized student registration system in order to accomplish this task. This corresponds well to the position of software buyers, who have a great deal of knowledge about the problem domain but little knowledge of what a computerized system must do to solve their problems.

Test Instrument and Measurement Technique

The test instrument consisted of a collection of subtests, three of which were appropriate for use in this data analysis. Excerpts from the three subtests are given in Appendix B. In the first subtest, called "Functions Expected", the seven functions expected of the student registration system were listed. Participants were tested on their ability to ascertain whether or not the system performed the desired functions.

For each question, two measures were calculated: understanding and misinterpretation. Assume for a question in subtest 1 that the function was, in fact, performed by the system so the correct answer was "I am certain this function is performed." The following relationship exists between points awarded for the understanding and misinterpretation measures:

Response	Understanding	Misinterpretation
I am certain this function is not performed.	0	2
I am fairly sure this function is not performed.	0	1
I don't know.	0	0
I am fairly sure this function is performed.	1	0
I am certain this function is performed.	2	0

The understanding score for each question ranged from 0, indicating no understanding, to 2, indicating high understanding. In this example the expected function was performed by the system so if the participant answered "I am certain this function is performed.", 2 points were awarded. If the participant answered "I am fairly sure this function is performed.", 1 point was awarded. Otherwise 0 points were awarded. Thus, in measuring understanding, points were awarded only if the participant was either fairly sure or certain of the correct answer.

The misinterpretation score for a question ranged from 0, indicating no misinterpretation, to 2, indicating a high degree of misinterpretation. In this example the expected function was performed by the system so if the participant answered "I am certain this function is not performed.", 2 points were awarded. If the participant answered "I am fairly sure this function

is not performed.", 1 point was awarded. Otherwise 0 points were awarded. Thus, in measuring misinterpretation, points were awarded only if the participant was either fairly sure or certain about an answer and that answer was incorrect.

The second subtest, called "Command Inferences", contained three problem scenarios for which more than one system command must be used (one of the problem scenarios is in Appendix B). Alternative solutions were presented and the participant was asked to rate the accuracy of each solution. The participant's response to each alternative solution was scored similarly to the questions in subtest 1 discussed above. In the third subtest, called "Data Item Sequencing", the participant was presented with 4 questions which examined the participant's knowledge about the information contained in data items at specific points during system execution (an example of these questions is in Appendix B). Each response was scored in the same manner as subtests 1 and 2.

The score for each of the two measures, understanding and misinterpretation, was the sum of the scores of the 38 (7+15+16) individual items. Because a maximum of two points was awarded for each question, the maximum understanding score and the maximum misinterpretation score were each 76.

RESULTS

The results for the measure of understanding are given in Figure 1. The F-statistic for Language X Expertise was significant at the .053 level, indicating an interaction between language and expertise. Examination of the four individual cell means illustrates the source of the interaction. There was no language effect on understanding for nonprogrammers but there was a language effect for programmers. Means for nonprogrammers (with a higher score indicating greater understanding) were 32.9 and 32.1 for structured English and Structured Analysis respectively, which is not significantly different. However, means for programmers were 52.4 and 41.7 for structured English and Structured Analysis respectively, indicating that programmers' understanding is significantly enhanced using structured English.

In sum, Structured Analysis and structured English were equally difficult for nonprogrammers to understand. For programmers, structured English resulted in significantly greater understanding than did Structured Analysis.

The results for the measure of misinterpretation are given in Figure 2. The F-statistic for Language X Expertise was significant at the .056 level, indicating an interaction between language and expertise. Again, examination of the four individual cell means illustrates the source of the interaction. There was a language effect for both programmers and nonprogrammers, but in opposite directions for each group. Nonprogrammers who read structured English had more misinterpretations than did nonprogrammers who read Structured Analysis. Means (with a high score indicating greater misinterpretation) were 15.5 and 12.9 for structured English and Structured Analysis, respectively. However, programmers who read structured English had fewer misinterpretations than did programmers who read Structured Analysis. Means were 12.2 and 14.4 for structured English and Structured Analysis, respectively.

In sum, for nonprogrammers, structured English resulted in significantly more misinterpretations than Structured Analysis. For programmers, structured English resulted in significantly fewer misinterpretations than did Structured Analysis.

Analysis of Variance

Source	DF	SS	MS	F	SIGNIF
Language	1	623.6	623.6	5.2	.026
Expertise	1	4172.7	4172.7	34.7	.001
Language X Expertise	1	464.7	464.7	3.9	.053
Error	73	8771.3	120.2		
Total	76	4076.7	185.2		

Cell Means

	Nonprogrammers	Programmers	Row Averages
Structured Analysis	32.1	41.7	36.7
Structured English	32.9	52.4	42.6
Column Averages	32.5	47.3	39.8

Figure 1 : Effects of Language and Expertise on Understanding

Analysis of Variance

Source	DF	SS	MS	F	SIGNIF
Language	1	0.7	0.7	0.3	.876
Expertise	1	19.1	19.1	0.7	.412
Language X Expertise	1	105.7	105.7	3.8	.056
Error	73	2051.4	28.1		
Total	76	2176.8	28.6		

Cell Means

	Nonprogrammers	Programmers	Row Averages
Structured Analysis	12.9	14.4	13.7
Structured English	15.5	12.2	13.8
Column Averages	14.2	13.2	13.7

Figure 2 : Effects of Language and Expertise on Misinterpretation

LEGEND DF = degrees of freedom; SS = sum of squares; MS = mean square; F = F- statistic; SIGNIF = significance level of F statistic

Combining both understanding and misinterpretation results, for programmers, structured English was both easier to understand and resulted in fewer misinterpretations than did Structured Analysis. For nonprogrammers, the languages were equally difficult to understand, but structured English resulted in more misinterpretations than did Structured Analysis.

DISCUSSION

The aforementioned results, considered together, are graphically portrayed in Figure 3. This study suggests that it is the rules of construction applied by the reader of a specification, in addition to the formality or informality of the language in which the specification is written, which influence both understanding and misinterpretation. Greater understanding will result if the reader of a specification knows the rules of the language in which the specification is written. Significantly more misinterpretations occur when a specification is written in a language using particular rules of construction but is read as if it were written using different rules of construction. Thus, ambiguity of a specification may not be solely a function of the degree of formality of the specification, but also of the way in which the specification is read.

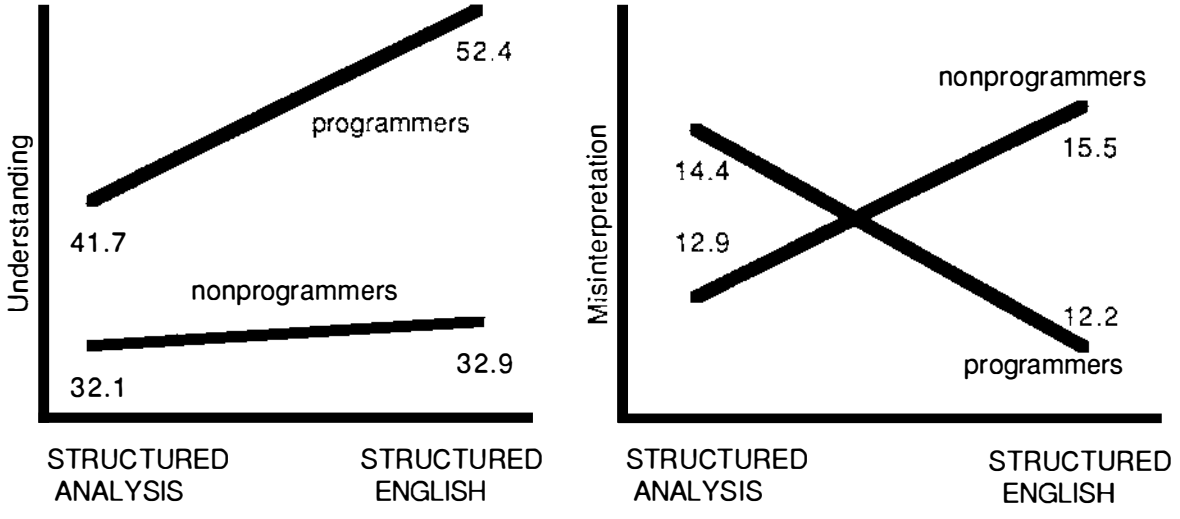


Figure 3 : Effects of Language and Expertise on Requirements Specification Understandability

Consistent with the above interpretation and with the results of this study is the hypothesis that programmers read structured English as a computer program and nonprogrammers read it as technical English [2]. The structured English specification was written using the formal rules of the construction of code. Nonprogrammers could not read it according to the formal rules with which it was written, as indicated by their low understanding scores. Due to its similarity to technical English, it was likely read using the rules of construction of technical English. This alternative perception of the structured English specification could be resulting in more misinterpretations of it. Nonprogrammers are unlikely to have rules of construction for data flow diagrams, resulting in low understanding scores. However, in contrast to structured English, nonprogrammers are also unlikely to have alternative rules of construction for data flow diagrams, resulting in fewer misinterpretations of them.

A parallel result occurs with programmers. Programmers reading structured English have greater understanding and fewer misinterpretations because the they know how to read code, and

the specification was constructed in this fashion. Because the programmers in this study were not familiar with data flow diagrams, they did not know the formal rules of their construction and how they should be read, thus programmers' lower degree of understanding of Structured Analysis. Given the similarity of data flow diagrams to flowcharts, it is possible that programmers read the data flow diagrams as if they showed flow of control rather than flow of data. The greater number of misinterpretations by programmers reading Structured Analysis could be a function of their being read as if they were flowcharts.

CONCLUSIONS

The results of this study suggest that it is how the reader interprets the specification as well as the formality or informality of the specification itself which determines understanding and misinterpretation. Thus, neither Structured Analysis nor structured English is ideal. If the possibility of misinterpretation of the requirements specification by a user is important in one's particular situation, Structured Analysis may be the better choice. It results in significantly fewer misinterpretations, but not significantly lower understanding on the part of nonprogrammers.

Structured English is the better choice for programmers who do not know Structured Analysis. They have significantly greater understanding and significantly fewer misinterpretations using structured English. However, programmers can be trained to read Structured Analysis. Although this study offers no experimental evidence as to how programmers who know how to read data flow diagrams would perform, if the interpretation of the results presented above is correct, understanding should be increased and the number of misinterpretations reduced.

This study was able to address the effects of language and expertise on misinterpretation only because a method of measurement was developed that experimentally differentiates between not knowing and misinterpreting. This measurement technique can be used to address a wide variety of issues in software engineering, as understandability is an important software quality characteristic during all phases of the software life-cycle.

REFERENCES

[1] Meyer, B., "On Formalism in Specifications," IEEE Software, Vol. 2, No. 1, January 1985, pp. 6-26.

[2] Davis, A.M., "The Design of a Family of Application-Oriented Requirements Languages," Computer, Vol. 15, No. 5, May 1982, pp. 21-28.

[3] Casey, B.E. and B.J. Taylor, "Writing Requirements in English: A Natural Alternative," IEEE Software Engineering Standards Applications Workshop, Aug. 1981, pp. 95-101.

[4] Fairley, R., Software Engineering Concepts, Chapter 2, "Planning a Software Project," McGraw-Hill, 1985.

[5] Pressman, R.S., Software Engineering: A Practitioner's Approach, Chapter 5, "Software Requirements Analysis," McGraw-Hill, 1982.

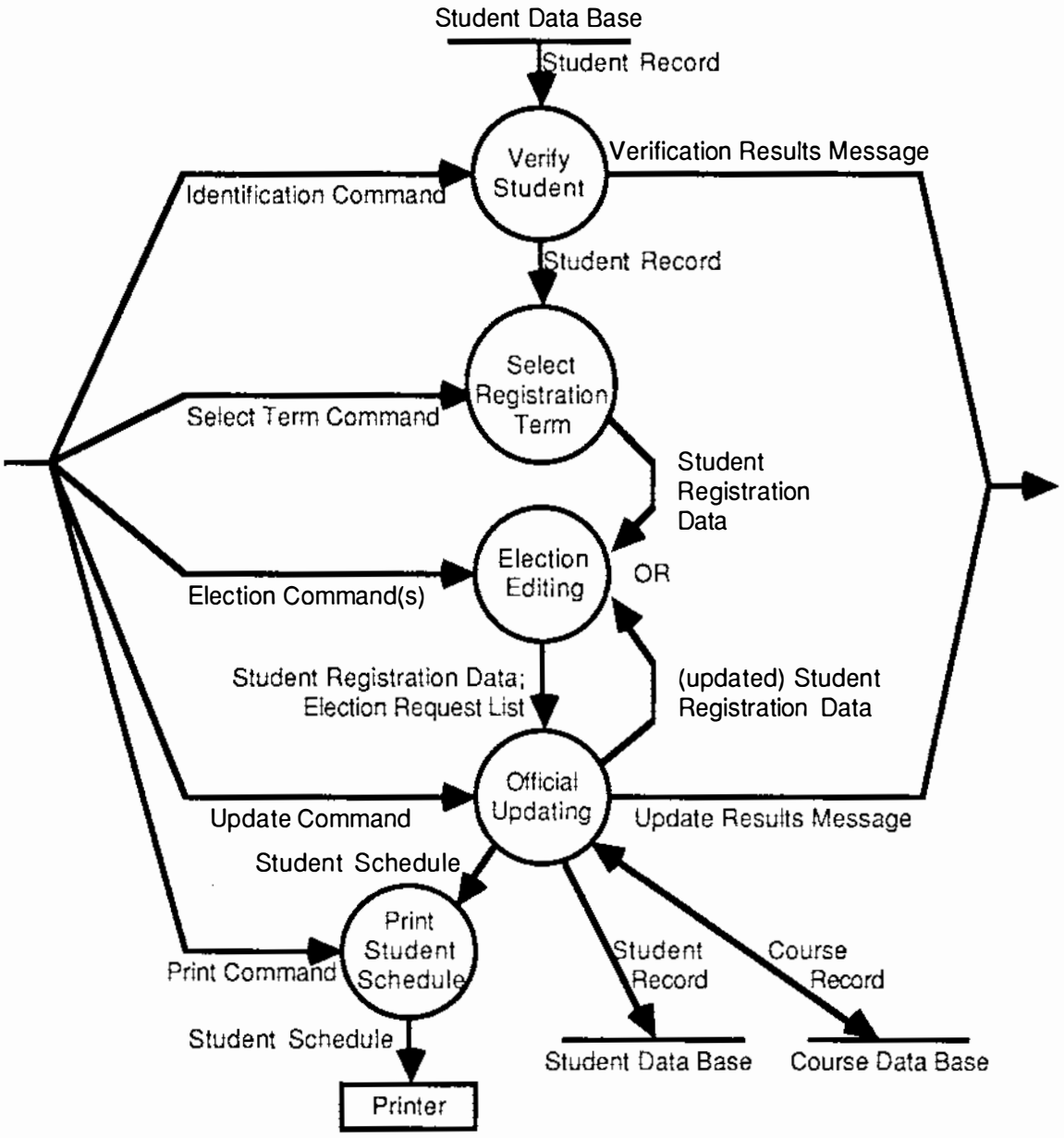
[6] DeMarco, T., Structured Analysis and System Specification, Chapter 1, "The Meaning of Structured Analysis," Prentice-Hall, 1979.

APPENDIX A

Register-A-Student [structured English]

Verify-Student.
If the student passes verification,
  For each term that the student wants to register for do the following:
    Select-Registration-Term.
    Repeat the following:
      Election-Editing.
      Official-Updating.
    Until the operator enters a Print-Command.
    Print-Student-Schedule.

Register-A-Student [Structured Analysis]



APPENDIX B

Excerpt From Subtest 1

Reproduced below is a list of the functions that the system is expected to perform. For each of the functions, indicate whether or not the system performs the function. Your response should be a number from 1 to 5.

1 = I am certain this function is not performed
2 = I am fairly sure this function is not performed
3 = I don't know
4 = I am fairly sure this function is performed
5 = I am certain this function is performed

1. Enforce size and entry restrictions on course enrollment.
2. Prohibit registration of students with an academic hold.
3. Print the student's class schedule at the end of the registration session.

Excerpt from Subtest 2

For each of the following questions please respond to each of the proposed answers. Your response to each answer should be a number from 1 to 5, indicating your view of the answer's accuracy.

1 = I am certain this answer is wrong
2 = I am fairly sure this answer is wrong
3 = I don't know
4 = I am fairly sure this answer is correct
5 = I am certain this answer is correct

1. Andy Haney just got a part-time job as a computer operator for the student registration system. Unfortunately, Andy's first day at his new job did not run as smoothly as he had hoped. While registering students, Andy had a tendency to forget to enter the Update-Command. What kind of behavior would the system exhibit when Andy did this?
   a. The student will not be officially enrolled in any classes but the student's complete schedule will be printed, making the student believe (wrongly) that he is enrolled.
   b. The registration system will become improperly synchronized and will print the schedule of the student who registered before this student.
   c. The system will not print the student's schedule.
   d. Forgetting to enter the Update-Command is not detrimental to system performance.
   e. The printed schedule of the student will be empty but the student will be enrolled in classes.

Excerpt from Subtest 3

For each of the following questions please respond to each of the proposed answers. Your response to each answer should be a number from 1 to 5, indicating your view of the answer's accuracy.

1 = I am certain this answer is wrong
2 = I am fairly sure this answer is wrong
3 = I don't know
4 = I am fairly sure this answer is correct
5 = I am certain this answer is correct

1. Suppose the system has just finished performing Election-Editing. What information is included in the Student-Schedule at this point in time?
   a. All the courses the student has registered for during previous terms.
   b. All the courses the student has registered for during previous registration sessions for this term.
   c. It is possible that there is no information in it.
   d. All of the pending course requests made by the student.

Specifications and Programs: Prospects for Automated Consistency Checking

Albert L. Baker[1] John T. Rose

Department of Computer Science
Iowa State University
Ames, Iowa 50011
(515) 294-4377
csnet: baker@atanasoff.cs.iastate.edu

Keywords:

specifications,
abstractions of programs,
software validation

Biographical Sketches

Dr. Baker received his B.A. Degree in Mathematics from Drake University, Des Moines, Iowa, in 1974 and his M.S. and Ph.D. degrees from The Ohio State University, Columbus, Ohio, in 1976 and 1979, respectively. He is currently an assistant professor in the Department of Computer Science at Iowa State University. His professional experience includes serving as Vice President for Operations for a communications research company and he is currently President of IRIS Systems, Inc., a supplier of natural language text analysis software. Dr. Baker's research interests are in specification languages, the structure of software, software verification and validation, and natural language text analysis.

Mr. Rose received a B.S. Degree in Psychology in 1982 and a B.S. Degree in Computer Science in 1985 from Iowa State University. Currently he is a graduate student in the Department of Computer Science. His current research interests include software specification, software verification, algorithm analysis, and systolic architectures.

[1]Dr. Baker will handle the correspondence on the paper.

1 Introduction

There is a deficiency common to a number of the widely used approaches to software development – it is, in practice too difficult and costly to maintain consistency between the documents that are produced. The structured analysis approach using dataflow diagrams as an initial specification document is typical. Most developers find value in using dataflow diagrams and data dictionaries to get an initial perspective on a complex system. However, once the dataflow perspective is completed, it is difficult, and maybe not cost effective, to maintain the dataflow specification as the system is designed and implemented.

In this paper we consider the prospects for automated support for a strong notion of consistency. We are concerned with more than just linking pieces of specifications with pieces of code. We speculate on the prospects for automated help in checking that what the implementation actually does is consistent with what the specification says it should do.

There are two premises fundamental to this approach:

1. The specification must be a formal description of the functionality of the software system. This is consistent with the current trend toward formalism in software specification techniques.

2. While there is still constant and rapid evolution of specification techniques, specifications are distinct from production programs. The language used and the level of abstraction usually distinguishes specifications and programs. Stated using a specific example, the main procedure of an Ada program is not a specification.

We use an abstract model approach to formal specifications in a language called SPECS. SPECS is analogous to other abstract model specification languages like Z [1], VDM [7], and Alphard [9]. These languages provide a primitive set of abstract objects which can be composed to define the domain of an abstract data type. Operations on the abstract types are specified using a well- defined language for pre and post condition assertions. The approach to consistency checking between SPECS and programs which we are considering is based on a dataflow perspective of both programs and specifications. An overview of the process is depicted in Figure 1. Each procedure or function in a program is viewed abstractly as a definition (assignment of value) of certain parameters and global objects in terms of other parameters and globals. This is consistent with traditional unit level dataflow analysis [5] in that ultimately the role of local variables is abstracted out. Each possible definition is predicated on values of another set of parameters and globals. This *abstract procedural definition* has been carefully defined using SPECS.

The abstract procedural definition for a given procedure or function is still given in terms of the data structures and variables of the program. The SPECS for the operation that the

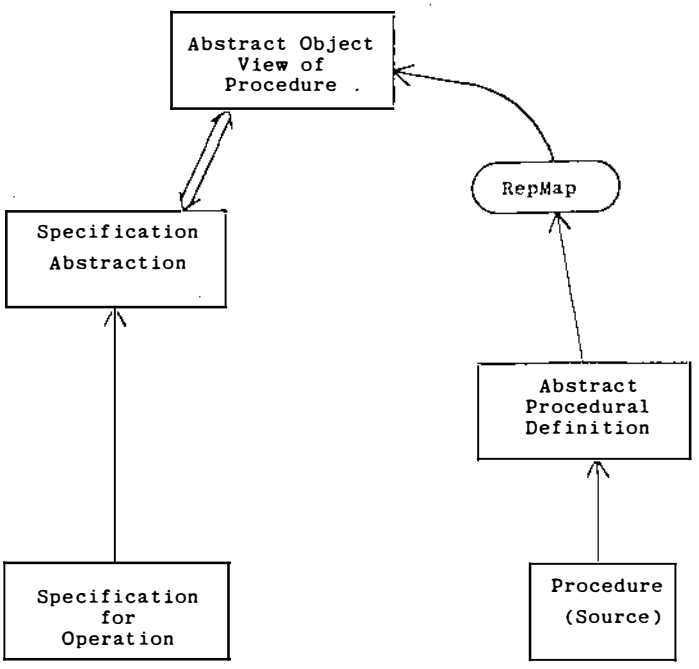


Figure 1: Consistency Check Overview

procedure or function implements are stated in terms of abstract objects. A representation mapping [6] from programming language data structures and variables to abstract objects provides an abstract object view of the procedure or function.

In order to see if the abstract object view of the procedure or function is consistent with the SPECS, we abstract from the SPECS an analogous functional view of the specification of the operation – what "new" abstract objects are produced using what other abstract objects, and on what abstract objects is each such definition predicated. It is this *specification abstraction* that we compare with the abstract object view of the procedure.

The remainder of the paper considers this approach to consistency checking in more detail and provides a complete example. In Section 2 we briefly describe the SPECS language and specify an abstract data type (ADT) BoundedSet. In order to avoid a programming language dependent description of consistency checking, we base abstract procedural definitions on a standard representation of imperative language programs [3]. This standard representation allows independence from particular languages and facilitates precise definitions. The standard representation is described in Section 3. In Section 4 we introduce our formal definition of *abstract procedural definition* and develop an abstract procedural definition for a procedure Insert which is part of an implementation of the ADT BoundedSet. In Section 5 we discuss prospects for automated consistency checking using the SPECS definition of Insert and the abstract procedural definition of the procedure Insert. In the concluding Section 6 we enumerate

both the pitfalls and prospects for the described approach.

2 A Development Paradigm Using Abstract Model Specifications

The approach to consistency checking explored in this paper is based on a particular object-oriented paradigm for software development. The SPECS language supports the formal definition of ADT's. In the usual manner, an ADT consists of a domain of objects and the specification of a set of operations on the domain. While a thorough understanding of the SPECS language is not required for this paper, a brief introduction is provided. A more comprehensive development of SPECS may be found in [2].

A SPECS domain is specified in two parts. The source set for the domain defines the structure of the objects in the domain. The primitive abstract types *set, sequence,* and *n-tuple* and an appropriate primitive set of abstract scalar types are used to compose the source set. SPECS uses a Pascal-like type declaration syntax for source set composition. Not all objects with the structure defined in the source set may properly belong in the domain. For example, if we model rational numbers as ordered pairs (2-tuples) of integers (Numerator, Denominator), then objects with Denominator = 0 are excluded from the domain. These restrictions are called *invariants* and are expressed using the First Order Predicate Calculus (FOPC) over the abstract source set.

Operations on the domain are defined in three parts. A Pascal-like header is used to name the operation and define input/ output parameters and return value types. It is important to keep in mind that in SPECS these types for parameters and return values are abstract types. The effect of operations are specified using pre and post conditions, expressed in FOPC. The pre condition asserts the conditions (state) which must hold in order for the operation to be applied. The post condition is a static assertion about the effect of the operation.

The SPECS for the example ADT BoundedSet which we will carry throughout the paper follows:

```
ADT:BoundedSet
    domain_A
        source set_A
            BoundedSet = set of E;
            E = generic;
        invariant_A
            for any S of type BoundedSet:
                0 ≤ |S| ≤ MaxSize
```

```
definitions_A
    constants_A
        MaxSize = any:integer;
operations_A
    :
    procedure Insert(var S: BoundedSet; X: E)
        pre_A : |S ∪ {X}| < MaxSize
        post_A: S = S' ∪ {X}
    :
```

The operation Insert will be used in our discussion of the prospects for automated consistency checking. Note that SPECS uses the prime notation ("'") on the abstract "variable parameter" S to denote the state (value) of S prior to application of the operation.

The first step in designing an implementation of an abstract model specification is selecting an appropriate implementation data structure for the abstract type. It is critically important that we describe the correspondence between the chosen data structures in the concrete (programming) domain and the abstract types. The chosen data structure must be adequate for representing all the objects in the abstract domain and each data structure value must unambiguously represent a unique abstract object. SPECS supports the formal statement of this mapping between concrete objects and abstract objects in a special ***specification function***[2] called the RepMap, for "representation mapping." The representation mapping rigorously defines, for each possible value of the chosen data structure, which object in the abstract domain is represented.

The following is one possible concrete domain for BoundedSet expressed in the programming language Pascal:

```
{ADT:BoundedSet}
    {domain_C}
        type
            BoundedSet = record
                             Vec: array[1..MaxSize] of E;
                             Last: 0..MaxSize
                         end;
            E    generic;
```

[2] A specification function in SPECS can be viewed as simply a notational convenience to help modularize complex assertions. Thus a specification function is just a parameterized name for a piece of a larger assertion.

```
    {invariant_C}
        {for any S of type BoundedSet:}
            ∀k[1≤k≤S.Last ⇒ S.Vec[k] "is defined"] ∧ Sorted(S)
            {Note that Sorted is an example of specification function in the concrete domain.}
{mapping
    function RepMap(S:BoundedSet_C): BoundedSet_A;
        RepMap = {c | IsInVec(S,c)}}
        {IsInVec is a true assertion if c is an array element value of S.Vec. }
{definitions}
    constants
        MaxSize = any:integer;
{Implementation of the operations: }
    ⋮
    procedure Insert(var S:BoundedSet; X : E);
    var i, Loc : integer;
    begin
        BinarySearch(S,X,Loc);
        {BinarySearch sets Loc equal to the index position
        in S where X should be inserted if it does not already exist
        and to the index position in S where S.Vec[Loc] = X otherwise.}
        if Cor(Loc = S.Last + 1, S.Vec[Loc] ≠ X)
            {Cor is our abbreviation for "conditional or"; not standard Pascal.}
            then
                begin
                    i:=S.Last;
                    while Loc<i do
                        begin
                            S.Vec[i + 1]:=S.Vec[i];
                            i:=i - 1
                        end;
                    S.Vec[Loc]:= X;
                    S.Last:=S.Last + 1
                end
    end;
    ⋮
```

3 A Standard Representation of Programs

As indicated in the Introduction, basing the definition of abstract procedural definitions on a standard representation of imperative language programs provides both rigor and language independence. The standard representation contains sufficient information to perform control flow, dataflow, and expression structure analysis. A SPECS-like definition of the standard representation used by the authors may be found in [3]. A careful definition of a mapping from the ISO Standard Pascal language to the standard representation may be found in [4].

While a detailed development of the standard representation is beyond the scope of this paper, detailed familiarity with the standard representation is only required for the formal definition of abstract procedural definition represented in [8].

The standard representation of a program consists of flowgraph-like representations of the individual procedures and functions contained in the program. Each such *UnitFlowStructure* is based in the usual notion of flowgraph in which each node of the graph represents a basic block of code, i.e., a fragment of source code which is always executed sequentially. Each basic block can consist of a sequence of assignment and procedure statements and a predicate.

4 Abstract Procedural Definition

As indicated previously, we suggest a dataflow view of production programs for automated consistency checking. We formally define an abstract procedural definition as an abstract object using SPECS. The definition is based on the standard representation of imperative language programs described in Section 3. In particular, we define an abstract procedural definition for each program unit. The definition characterizes which formal parameters and nonlocal variables may be assigned values (defined) as a result of the units' execution. It also characterizes the values that may be referenced in each of these definitions. Informally, an abstract procedural definition provides a view of individual program units as though they were concurrent assignment statements. The specification of an abstract procedural definition presented in this paper is well-defined for programs without recursive procedures or functions. It accounts for all the explicit control structures and for the full range of data structures.

The outermost levels of the formal specification of abstract procedural definitions are presented in this paper. A copy of the complete formal specification is available as [8].

```
ADT:Abstract Procedural Definition
domain
    source set
        AbstractProceduralDefinition -- set of PathDefinitions;
```

```
PathDefinitions = set of VariableDefinitions;
VariableDefinitions = triple(
            DefinedVariable:VarID,
            ReferencedVariables:set of ExprComponent,
            PredicateVariables:set of ExprComponent);
ExpType = sequence of ExprComponent;
ExprComponent = VarID|ConstID|FunctionUse;
FunctionUse = ordered pair(
          FunctionName:UnitID,
          ActualParams:sequence of ExpType);
```

VarID, ConstID, and UnitID are simply identification tags given to source code variables, constants, and program units in the standard representation.

Informally, a VariableDefinition contains three pieces of information. The DefinedVariable field represents a parameter that would be defined as a result of a procedure invocation. ReferencedVariables represents the set of variables, constants, and function uses which are referenced, either directly or indirectly, in the definition of the DefinedVariable. These referenced variables are either parameters or global variables. PredicateVariables represents the set of variables, constants, and function uses which predicate, directly or indirectly, the particular definition.

Each possible execution path through a procedure will result in the definition of particular parameters and global variables. Each such assignment is represented as a VariableDefinition. A variable parameter, i.e., reference parameter, that is not explicitly defined along a given path is considered to be defined by its original value. The set of VariableDefinitions for a particular path makes up a PathDefinition. Thus an abstract procedural definition is a set of PathDefinitions, where the PathDefinition for each path through the program unit is an element of the abstract procedural definition. Although we specify an abstract procedural definition in terms of "all paths", the size of this set is bounded. (Since we are only concerned with which variables are defined in terms of which other variables, it follows that there are only a finite number of possible execution paths that can yield distinct combinations.)

Even though we are just at the stage of considering the prospects for automated consistency checking, we have tried to insure that our observations are based on precise definitions. For example, we have defined a specification function that, given the standard representation of a procedure, defines the corresponding AbstractProceduralDefinition.

However, consideration of the procedure Insert presented in Section 2 provides good intuitive understanding of our approach. The simple path through the procedure which is taken if X is already one of the array element values (when Loc $\neq$ S.Last + 1 and S.Vec[Loc] $-$ X) results in S not being redefined and this path is predicated by the parameters S and X and by the

constant 1. What follows is the AbstractProceduralDefinition for the procedure Insert and the first item in the set reflects the simple path just discussed:

```
{ {(S,{},{S,X,1})},
   {(S,{},{S,X,1,2})},
   {(S,{S,X,1},{S,X,1})},
   {(S,{S,X,1},{S,X,1,2})},
   {(S,{S,X,1,2},{S,X,1,2})}
}.
```

This AbstractProceduralDefinition can be further abstracted. On the one hand S may not change based on a predicate we can call ϕ. In all other cases S is defined as a function of the original S' and X based on a predicate ϕ' of S' and X. Thus our abstract view of the procedure can be expressed as $(\phi(S',X) \; ? \; S \quad S') \wedge (\phi'(S',X) \Rightarrow S = \theta(S',X))$. This can be viewed as saying simply that some relation between S and X results in the program unit not explicitly defining S, and some other relation between S and X results in S being defined in terms of the initial values of S and X.

A flowgraph representation of the Insert code is given along with an annotated version of the abstract procedural definition of Insert in [8]. This shows more clearly how the abstract procedural definition is derived from the standard representation. In the next section we explore possibilities for consistency checking based on the Insert example.

5 Possibilities for Consistency Checking

Our goal is to automatically analyze consistency between an implementation, which is represented by an abstract procedural definition, and its corresponding abstract model specification. Recall that an abstract procedural definition is an abstract view of objects, and their relations with each other, in the concrete (implementation) world. But an abstract model specification document views objects in the abstract world. Hence, a bridge is necessary to know which abstract object is represented by a concrete object. This is one role of the representation mapping described in Section 2. In our example, the abstract BoundedSet represented by a particular value of the Pascal record variable S simply contains the values of the elements of the array S.Vec subscripted from 1 to S.Last. Since this representation mapping is straightforward in our example, in the rest of the discussion we do not distinguish an abstract BoundedSet S and a Pascal variable S of type BoundedSet.

There are some easily observed consistencies in our example. The BoundedSet S is the only object defined in the abstract specification of Insert and it is also the only object defined in the

abstract procedural definition of Insert. Also, the abstract post condition for Insert, $S = S' \cup \{X\}$, suggests that the objects which should define S are S′ and X. These two objects appear in the ReferencedVariables set in three of the five PathDefinitions in the abstract procedural definition of Insert. So at least some paths are consistent with the basic form of the abstract post condition. We next explore three levels of consistency checking which we feel can be automated. Our discussion of each level is based on the BoundedSet example.

Consider the following three levels of consistency:

1. *Interface consistency* deals with the question of what gets defined by a program unit. Do only the specified objects get defined?

2. *Functional consistency* means that the defined objects are defined in terms of other appropriate objects.

3. *Predication consistency* occurs when the objects which predicate the occurrence of a definition are consistent between documents.

The most obvious consistency in the BoundedSet example is at the interface level. As mentioned earlier, it is clear from the abstract specification that only the BoundedSet S should be modified by the Insert procedure, since only S is defined in the post condition for Insert. It is also clear from the abstract procedural definition of Insert that the code is consistent with the specification in this respect because only the object S is defined.

In large scale software projects perhaps the most common errors occur at the interface level. This is the level of consistency checking that is straight-forward to implement and may even detect the largest number of errors. Imagine a large software project where in a procedure a global variable is modified and this change is not reflected in the specification. In other respects the procedure works "correctly". An error may occur in another procedure which indirectly calls this one. This type of error is frequently difficult to isolate. The errant procedure may pass unit tests and the error may not be discovered until system integration testing is done. The abstract procedural definition would identify this global definition as an inconsistency between the code and the specification document prior to any testing.

We mentioned earlier that three of the five PathDefinitions in the abstract procedural definition of Insert were consistent with the specification in the sense that S was being defined in terms of the appropriate objects, S′ and X. How can we explain the other two PathDefinitions? This simple example demonstrates that the resolution of functional consistencies, and, as we shall see shortly, predication consistencies, is a harder problem than resolving interface consistencies. Before returning to the explanation of the two PathDefinitions where S is not explicitly defined, we consider possible transformations based on abstract model specifications.

Recall from Figure 1 that some transformation of the specification document is suggested. The need for this transformation is to make what is clear to a human reader (the semantics of

the specification) clear to an automated consistency checker. Consider again the abstract post condition for the Insert operation. Is it clear that in some cases S will not be changed? From our mathematical understanding of set union we know that if $X \in S$ then S will be unchanged. Is it clear what the predicates are? Consider now an equivalent post condition for the Insert operation, $(X \in S \Rightarrow S = S') \wedge (X \notin S \Rightarrow S = S' \cup \{X\})$. From this form of the abstract post condition, it is clear there are cases in which S is not changed by Insert. It is also clear that S and X predicate whether S is actually changed. Thus we bump up against the fundamental problem of most automated verification systems — which transformations do we apply? In our case, the heuristics can be based on the form of the abstract object view of an abstract procedural definition.

Consider again the two unexplained PathDefinitions in the BoundedSet example. These PathDefinitions leave S unmodified by the Insert operation. These paths are consistent with the abstract specification in the case where $X \in S$. Note that this consistency does not mean that the defined objects are, in fact, defined correctly. It does give us a useful check on the actual behavior of the procedure with respect to the specification.

What can we say about predication consistency in the BoundedSet example? From the original abstract post condition the predicates are not very clear. However, after transforming the abstract post condition into a conjunction of implications the predicates are apparent. Since S and X are found to predicate all definitions of S in both the specification (after transformation) and the abstract procedural definition, predication consistency is established. Had we found any path through Insert which was not predicated by both S and X or was predicated by some other non-constant object an inconsistency would be identified. Both predication and functional consistency checking are sensitive to the form of the abstract specification. Thus heuristics for transformations of abstract specifications are essential.

6 Conclusions

We have been sufficiently encouraged by the prospects for automated consistency checking between abstract model specifications and source programs to continue development of the heuristics and transformations necessary to do more than just interface consistency checking. We are also seeking opportunities to implement our formal definition (specification) of abstract procedural definition and to get this embedded in a larger software development environment.

While we are of course optimistic that this particular line of research will continue to yield useful results, we are firmly convinced that it is this more formal approach to software development and software development environments that affords the greatest prospect for significant gains in software development and maintenance productivity and reliability.

References

[1] J.R. Abrial and S.A. Schuman. Non-deterministic system specification. In G. Kahn, editor, *Semantics of Concurrent Computation*, pages 34-50, Springer Verlag, Lecture Notes in Computer Science, No. 70, 1979.

[2] A. L. Baker, J. M. Bieman, and P. N. Clites. Implications for formal specifications: results of specifying a software engineering tool. In *Proceedings of the Eleventh Annual International Computer Software & Applications Conference (COMPSAC-87)*, IEEE Computer Society and the Information Processing Society of Japan, Tokyo, Japan, October 1987.

[3] J. M. Bieman, A. L. Baker, P. N. Clites, D. A. Gustafson, and A. C. Melton. A standard representation of imperative language programs for data collection and software measures. *The Journal of Systems and Software*, 8(1):13-37, January 1988.

[4] K. Doh, J. M. Bieman, and A. L. Baker. *Generating a Standard Representation from Pascal Programs.* Technical Report TR 86-15, Department of Computer Science, Iowa State University, 1986.

[5] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *Computing Surveys*, 8(3), September 1976.

[6] J. Guttag. Abstract data types and the development of data structures. 20(6):396 404, June 1977.

[7] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 1986.

[8] J. Rose and A. L. Baker. *Formal Definition of "Abstract Procedural Definition": A Global Data Flow Perspective on Procedures.* Technical Report TR 88-16, Department of Computer Science, Iowa State University, 1988.

[9] M. Shaw. *Alphard: Form and Content.* Springer-Verlag, New York, 1981.

Design

"Fault-Tolerant Software and Object-Oriented Design"
Michel Bidoit and Christophe Dony, Laboratories de Marcoussis, Centre de Recherche de la CGE, France

"Validation by Pre-reviewing and Justification"
Ilkka Tervonen, University of Oulu, Finland

"Multi-Person Projects and CASE"
Byron Miller, KnowledgeWare, Inc.

Fault-Tolerant Software and Object-Oriented Design.

Michel Bidoit[1], Christophe Dony[2]

Abstract

The aim of our paper is to describe how the object oriented formalism can improve the expressive power of an exception handling system and how it can simplify its implementation. Moreover, we show that, although object-oriented design already proved to be successful for enhancing software reusability and extendibility, embedding an exception handling system within an object-oriented language further enhances reusability and software quality.

Biographical sketch

① Laboratoires de Marcoussis - Centre de Recherche de la CGE,
Route de Nozay, 91460 Marcoussis, France.
&
L.R.I., C.N.R.S. U.A. 410 "Al Khowarizmi"
Universite Paris-Sud - Bat. 490 - 91405 ORSAY Cedex, France
E-mail: mb@sun8.lri.fr

Michel Bidoit is a graduate of the Ecole Normale Superieure de Saint-Cloud. He received the "Agregation de Mathematique" degree in 1979 and the "These de 3eme Cycle" degree in Computer Science in 1981 from the University of Paris-Sud, Orsay, France. From 1980 to 1982 he was employed as a Researcher by the Laboratoires de Marcoussis, the Research Center of the Compagnie Generale d'Electricite (C.G.E.). He has joined the "Programmation and Software Engineering" Team of the Computer Science Laboratory (L.R.I.) of the University of Paris-Sud since 1982 and is currently a C.N.R.S. "Chargé de Recherche". He is also acting as a Consultant to the Laboratoires de Marcoussis. He is interested in most aspects of software engineering, more especially in algebraic specification techniques and the impact of exception handling, both at the specification and programming levels.

② Laboratoires de Marcoussis - Centre de Recherche de la CGE.
&
L.I.T.P. - Universite Paris VI, 4 place Jussieu, 75005 Paris, France.
E-Mail : dony@crcge1.cge.fr
Tel : (33 1) 64 49 10 03

Christophe Dony received a Masters Degree in Computer Science in 1985 from the University of Paris-VI, Paris, France. He has been working since 1985 as a Research Engineer at the Laboratoires de Marcoussis (C.G.E) and is completing a Ph.D. at the University of Paris-VI in the area of object-oriented languages and of interactive programming environments. His research interests include languages for AI, programming methodology, programming environments and software engineering.

Fault-Tolerant Software and Object-Oriented Design.

Michel Bidoit, Christophe Dony

Introduction

Exception handling techniques [Goodenough 75] [Levin 77] [Knudsen 87] have been largely studied in the past years, notably in order to allow the implementation of fault-tolerant programs, modules or encapsulations [Liskov 79] [Ichbiah 79] [Christian 82] [Bidoit 85]. These issues are important on several accounts for the software quality. For example, programs in execution are able to stay in a coherent state and sometimes able to react after an exceptional situation occurrence. Users can be notified when a module fails and failures can generally be located. Furthermore, defining fault-tolerant encapsulations enhances modularity. A fault-tolerant encapsulation can be defined as a software object performing some services (for example implementing an abstract data type) and hiding how it implements them. For that, it must be able to handle lower level exceptions raised by inner module activations, and to return well defined and foreseen answers, whatever could happen during its execution, even though an exceptional situation occurs. Any user can specify his own responses to these exceptional situations by defining appropriate error handlers. Dialogues based on exceptional situations are thus possible and the program reusability is improved. Finally, as far as all the possible responses of a module to its legal inputs can be specified in its interface, program readability is also improved.

Object oriented programming and design, mainly initiated by Simula [Dahl 70] and by Alan Kay works leading to *Smalltalk* [Goldberg, Kay 72] [Goldberg, Robson 83], is a software decomposition technique that has led to improvements in system specification and software quality. The object-oriented design brings an original and more natural way to specify a certain kind of problems (e.g. complex system simulation) through the description of their compound objects rather than through the specification of isolated procedures [Cox 83] [Booch 86]. Modularity and information hiding are achieved through object classes that realize data abstraction. Subclassing, inheritance and message sending are fundamental properties allowing sharing of code [Lieberman 86], reusability and extendibility [Meyer 87].

In this paper, we show how the expressive power of the object oriented formalism provides a promising set of solutions to many exception handling issues, such as creating user-defined exceptions, passing arguments from signalers to handlers through structured objects, signaling fatal or continuable exceptions with the same primitive, and the like. Inherent characteristics of object-oriented languages make it possible to introduce new facilities that cannot be simply implemented within procedural languages. For example, exceptions can be first class objects and can be organized in a hierarchy that will be taken into account by the handling mechanisms; slots (i.e. attributes) and dynamic properties can be defined on exceptions; all handling operations can be performed via message sending, which makes them generic. The key idea is not only to consider the data used by the program as abstract data types, as suggested by the object oriented approach, but also the exceptions as well, as proposed earlier in *Zetalisp* [Moon 83], *Taxis* [Nixon 83] or in reversible object-oriented interpreters [Lieberman

87]. This turns out to be a simple and elegant way to design a powerful and user-friendly exception handling system.

In a final section, we focus on the ability to specify and write reusable programs using an object oriented language. To design programs that should be both reusable and fault-tolerant is not an easy task, and we shall show how our exception handling system makes things easier.

The ideas described in this paper have been the basis of the specification and implementation of an exception handling system for the *Lore* object-oriented language[1] [Caseau 86], [Caseau 87]. However, the aim of this paper is not to discuss the exception handling model chosen for *Lore* (e.g. resumption or termination) neither the exception handling requirements that are specific to object-oriented programming (see e.g. [Dony 88]), but rather to focus on those points that are relevant for any object oriented language.

1. How far the object oriented approach may improve exception handling.

1.1. Terminology

• Object-oriented design.

Let us recall that object oriented programs allow designers to structure a software system around its compound objects instead of around a global functionality to be implemented. **Classes** of **objects** implements abstract data types on which operations (usually named **methods**) can be defined. **Instances** of classes are elements of the type; each one owns a particular value for each **slot** (slots can be thought of as attributes) defined on its class. **Inheritance** distinguishes object-oriented languages (OOL) from modular languages like *Ada*, since abstract data types (classes) can be organized in a hierarchy: each element of a type inherits the properties defined on its upper types (also called ancestors or **super-classes**). The communication protocol is based on message sending, which is an extension of overloading, taking into account inheritance. Message sending and inheritance are key-points for reusability and extendibility.

For each class C, methods (i.e. procedures) can be defined, each one being associated with an identifier (called **selector**). Message sending roughly works in the following way : sending the message M (M is the selector) to an object O of type C, i.e. belonging to the class C, entails the execution of the operation associated to M found in the smallest class containing O (which is either C or an ancestor of C). Thus, message sending can be thought of as an indirect procedure call, where the procedure to be executed is determined using the smallest class containing O where a method associated to the selector M has been defined. When such an operation can be found neither on C nor on its upper-classes, the exception **unknown-selector** is raised.

Within examples sketched in this paper, sending the message "selector" to the object "receiver" with some arguments is expressed by : "[receiver selector arg1 ... argn]".

• Exception handling.

Software failures reveal either programming errors or the application of correct programs to an ill-formed set of data; more generally, an **exception** can be defined as a situation leading to an impossibility of finishing an operation. The term "exception" implies that this situation is not always an error

[1]developed at the *Laboratoires de Marcoussis, CGE Research Center.*

case. Three types of exceptions can be distinguished (see [Goodenough 75]) : the **domain exceptions** raised when the input assertions of an operation are not verified, the **range exceptions** raised when the output assertions of an operation are not verified or will never be, and the **monitoring exceptions** raised to implement controlled non-local moves. The problem of handling exceptions is to provide materials and protocols allowing to establish a communication between a routine which detects an exceptional situation while performing an operation and those entities that asked for this operation (or have something to do with it).

An exceptional situation is detected when the usual sequence of statements has to be interrupted and an exceptional computation has to be executed. **Signaling** the exceptional situation leads to this interruption, followed by a search and an invocation of a **handler** for the particular exception. **Handlers** are attached to (or associated with) entities for one or several exceptions (according to the language, an entity may be a program, a process, a procedure, a statement, an expression, or a data). Handlers are invoked when an exception is signaled during the execution or the use of one of these protected entities (for further precisions on terminology and explanations on these concepts, see [Goodenough 75], [Liskov 79] or [Knudsen 87]).

A powerful exception handling system provides specific primitives and predefined protocols to define handlers, to signal exceptions, and to handle them.

A signaler has to identify the exceptional situation, to interrupt the usual sequencement, to look for a handler referencing the exception, to invoke it and to pass it relevant information about the exceptional situation, such as the context in which the situation arises. It is also useful for handlers to know whether the exception is fatal or whether the computation can be resumed after relevant corrections; if so, the signaler might want to predict which kind of correction are possible in this particular situation; indeed, resumption should not be achieved without the agreement of both the signaler and the handler.

Handlers have to set the system back to a coherent state, so that standard computation can start again. Various models have been defined to state what happens after an exception is raised: handlers may have to choose, knowing about the current context and using information provided by the signaler, whether to transfer control to the statement following the signaling one (resumption), or to discard the context between the signaling statement and the one to which the handler is attached (termination). Finally, before being invoked, handlers have to be associated with both specific entities (in order to protect a particular invocation of a given operation) and with more global ones (in order to provide **default-handlers**).

Let us now see how the object-oriented formalism allows to implement and to improve these functionalities in an efficient and simple way.

1.2. Status of exceptions.

A first issue is the status of exceptions; when an exception handling system is provided in a language, how are exceptions represented or referenced? How can they be manipulated or inspected?

• First class entities.

When a procedure invocation detects an exceptional situation, the issue is to find the piece of code relevant to this situation and which must be executed. In the most simple systems, a transfer of control toward a label or a specific procedure call is wired into the signaling routine. In most of the important exception handling systems that can be found in procedural languages (e.g. *PL/I*, *Ada*, *Clu*,

Mesa), exceptions are identifiers; when an exception is raised, a handler that references this identifier is looked for and invoked. A common characteristic of these systems is that knowledge relative to exceptions (even the most general one) is, in the first case, concentrated in a unique handler (implying problems to modify and update it), and in the second case, scattered in user-defined handlers and default handlers provided by the system, and is therefore uneasy to grasp. In both cases, exceptions are not first class objects but identifiers; subsequently they cannot own any characteristics, cannot be inspected, modified or enriched.

Exceptions are nevertheless complex entities, they own not only attributes such as the context where they are raised or the list of all their possible handlers, but also procedural characteristics that describe for example how to report an error message or how to propose some solutions to the failure. As soon as we represent exceptions as data types, following the ideas developed in the *flavors* system [Moon 83] or in *Taxis* [Nixon 83], it becomes simple and natural to specify their static characteristics as attributes (slots) and their dynamic knowledge as methods (procedures).

• User-defined exceptions.

In the languages where the types *exception* or *condition* are defined, it is generally possible for users to define new exceptions by declaring identifiers to be of that type. This allows to provide specific names to new exceptional situations raised in user programs, and to define relevant handlers for these situations. In our system, as each kind of exception is a class, the concept of exception is represented by a meta-class (a type able to generate types) named **exception-class**. Creating a new (kind of) exception can be done by creating a class, i.e by a simple instantiation of our specific meta-type. This operation is opened to all users.

Here is an example where new exceptions have to be defined. Consider the class *window*, where the slots *x-origin*, *y-origin*, *length* and *width* of range integer are defined; and the class *terminal*, where the slots *screen-length* and *screen-width* and the method *display-window* are defined. These classes could be part of a package allowing users to perform terminal independent video programming. Displaying a window on the screen can be performed by sending to the variable representing the current terminal the message *display-window* with an argument which is the window to be displayed. Assume that *display-window* signals an exception when a window is larger than the current terminal screen. It could be possible for that to raise a predefined high-level, general purpose exception. However, creating a specific new kind of exception leads to numerous advantages. The new exception can be created in the following way :

```
[exception-class new                    ; the protocol to instantiate classes
   name: window-larger-than-screen      ; the name of the new kind of exception
    ....]
```

The first advantage is that there is no difference between system and user defined exceptions, all can be created, raised and handled exactly in the same way. The other consequences lie in the automatic integration of the new user-defined exceptions into an inheritance lattice. Since the main interest of the exception lattice lies so far in property inheritance, let us detail what does it mean to define properties on exceptions and how this can be done.

• Properties defined on exceptions.

Two levels of knowledge about exceptional events are of first interest.

— **Slots.**

The former is made of all pieces of informations concerning a particular occurrence of an exception that have to be transmitted from signaler to handlers. When exceptions are identifiers there is no simple and predefined way to associate them knowledge and moreover no simple way to retrieve it. When exceptions are classes, a slot can be designed for each piece of information to be transmitted and message sending can be used to retrieve it.

For example, consider our exception *window-larger-than-screen* again, it is obvious that any handler for that exception is likely to make use of the characteristics of the window to be displayed. Here is the definition of a slot defined on the exception where this information can be stored.

```
[window-larger-than-screen has slot        ; syntax for defining a slot
   window-to-be-displayed                  ; its name
   range: window]                          ; its type
```

A second example is given by the most general exception of our system, named **exceptional-event**. Stating that any handler for any exception want to know where an exception has been raised, we define on *exceptional-event* two slots named **xprop** and **xobj** designed to store respectively the current property and the current receiver at signaling time. We shall explain later how handlers benefit from these slots in any cases.

```
[exception-class new  name: exceptional-event]
[exceptional-event has slot xobj  range: object]
[exceptional-event has slot xprop  range: property]
```

— **Methods.**

The latter level of knowledge concerning exceptions is made of all pieces of information designed to handle them. As we have said before, this kind of knowledge is usually, in non object-oriented exception handling systems, discarded in various handlers and not easy to grasp.

This looks right for local information, the extent of which is dynamic, as for example the knowledge embodied into a handler attached to a particular instruction (for example a handler saying that a particular call of the *divide* procedure should return 0 if *zero-divide* is raised). Such handlers are only relevant while their correlated instruction is being executed; subsequently, and since exceptions are global entities, it would not make much sense that these handlers were attached to the exceptions that they handle.

Besides, both the most general default handlers and the routines used within them are, by definition, valid regardless of any execution context. Instead of defining them as isolated pieces of code, the model that we choose allows users to specify any kind of default handlers as methods defined on exceptions.

Here is for example the method **handles-default** defined on *exceptional-event*, which is the most general default-handler, invoked after any exceptional event when no more specific handlers can be found. Each selector used in this method points out a property which is also defined on *exceptional-event* but can be redefined for each exception, as we shall see while talking about reusability.

```
[exceptional-event has method handles-default
    ... method's body...
    [oself[2] describe-exception]          ; Reports the exceptional event
    [oself describe-context]               ; Displays the context of the event
    [oself display-propositions]           ; Displays propositions for proceedings
    [oself return-to-top-level] ]          ; if no proposition is chosen.
```

A second example of methods defined on exceptions are specific properties called **propose-method** and designed to display some interactive propositions for resumptions. Of course, a method is also defined to perform the relevant actions if this proposition is chosen. For example, when the exception *window-larger-than-screen* is raised, the following proposition "Display the visible part of the window" is displayed.

All the properties defined on exceptions can be retreived by inspecting them, can themselves be inspected and can be invoked by message sending. Thus, inspecting an exception provides a good idea of what can happen when it is raised. The figure 1 shows a part of the information that is delivered while inspecting the exception *exceptional-event*.

Fig. 1 : Inspecting the root-exception: "exceptional-event".

```
Lore> [exceptional-event describe]
is-a: <exception-class>
name: exceptional-event
comment: "the root of exception's lattice."
superset: (<object>)
subset: (<warning> <error>)
dictionary: ...
genealogy: (<object> <lore>)
instances: nil
------
Slots defined on <exceptional-event>:
xhandler: <slot>, init-value: <unknown>, "The handler selected to handle the exception."
xprop: <slot>, init-value: <loop.top-level>, "The active property at signaling time."
xobj: <slot>, init-value: <unknown>, "The active object at signaling time."
xargs: <slot>, init-value <unknown>, "General arguments passed to handlers."
-----
Methods defined on <exceptional-event>:
handles-default: <method>, "System's default handler for all exceptions."
describe-context: <method>, "Display the context of the exception : active object , etc."
..... and so on
```

• Exception hierarchies.

Of course, the key idea which underlies the choice of designing exceptions as classes is to take advantage of the ability to organize them into a hierarchy[3] that will reflect common behavior, leading to reusability and extendibility. The object oriented problem decomposition ability fits well to express

[2]The identifier *oself* or *self* is a standard in OOP, it represents the receiver of the current message. Here, "[oself describe-exception]" means "send the message *describe-exception* to the instance of the exception for which *handles-default* has been called".

the relationship between all exceptions of a system. This means, for example, that the exception *zero-divide* can be implemented as a subconcept of the higher level exception *arithmetic-exception*. Similarly, the root of our exception lattice is divided (cf. fig. 1 & 2) into **fatal-event**, to which is attached the protocol for termination, and **proceedable-event** to which is attached the protocol for resumption[4]. Then **error** (cf. fig. 2) is the set of exceptional events for which resumption[5] is impossible; **warning** is the set of exceptional events for which the termination is impossible (i.e. resumption is mandatory); finally, multiple inheritance is used to create the class **exception** in order to allow both capabilities.

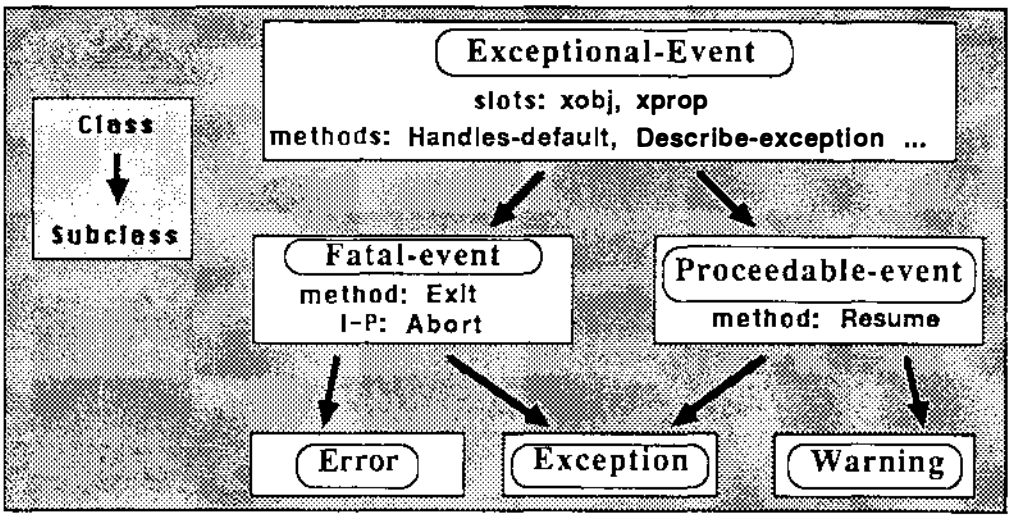


Fig. 2 : Root of the exception hierarchy.

System as well as user defined exceptions are appropriate sub-classes of these three main classes, depending on whether the exception is intended to be a fatal one or a resumable one.

• Inheritance and reusability.

Reusability and extendibility are the first consequences of this organization. Each new exception inherits, as soon as it is created, from the characteristics of its ancestors in the hierarchy. The above displayed method *handles-default* is thus included in the protocol of any exception; however, some parts of it can be overridden at each level to provide local behaviour. Thus, the method *describe-exception* is redefined on *window-larger-than-screen* so that, when this exception is raised, the invocation of the method *handles-default* produces :

*** The window W is larger than current terminal screen.
*** Exception signaled while sending message *display-window* to *current-terminal*.
1 : Abort.
2 : Inspect or modify W and retry operation.
3 : Display only the visible part of W, starting from left corner.
....

[3]This hierarchy should not be confused with hierarchy of handlers, as discussed in [Knudsen 87], that represent the ability for exceptions to be propagated from inner handlers to outer ones along the invocation chain.

[4]We choose to implement a resumption model where handlers still have the ability to perform termination.

This example reveals that the propositions for proceeding are also inherited. In order to highlight this point, let us complete our very simple "display" example to present a hierarchy of user-defined exceptions with embedded propositions (cf. Fig. 3). The very general "abort" proposition is defined on *fatal-event*. The second proposition, relevant for all exceptions raised while manipulating windows, is defined on *window-display-exception*, which is a subclass of *exception* and a super-class of *window-larger-than-screen*. *Window-display-exceptions* is an abstract[5] exception; the slot *window-to-be-displayed*, itself useful for any window relevant exception, is in fact defined on it, rather than on *window-larger-than-screen*.

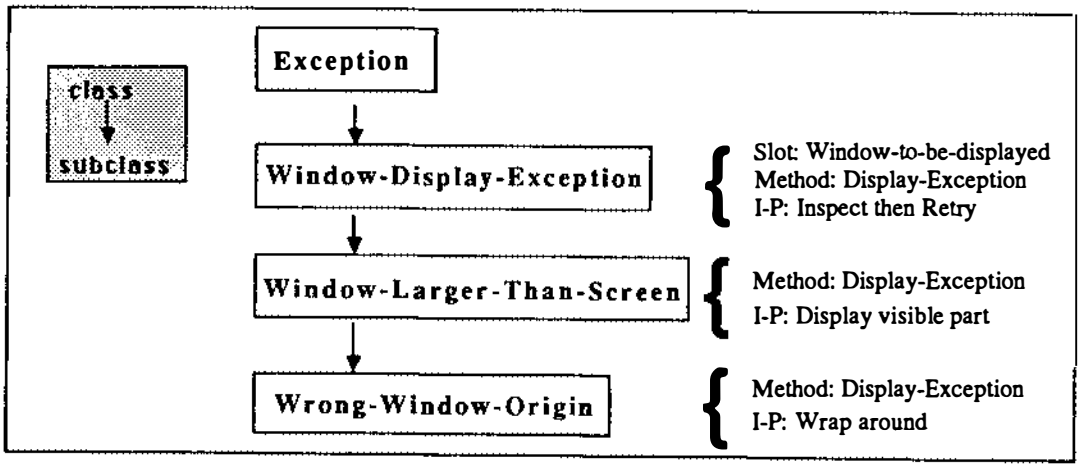


Fig 3 : A hierarchy of user-defined exceptions.

Finally, a last exception named *wrong-window-origin* is raised when a window comes out of the screen because its *x-origin* (resp. *y-origin*) added to its side length (resp. width) is higher than the screen length (resp. width). When this situation is about to happen, the second proposition is relevant since it allows to decrement the slot *x-origin* and the third proposition is obviously relevant too. A last possible solution could be to consider the screen to be circular and to display the right-end of the window on the left of the screen. Subsequently, our last exception can be created with benefit as a subclass of *window-larger-than-screen*. A method named *describe-exception* as well as a new propose-method named *wrap-around* implementing the last proposition for resumption are defined on it.

```
[wrong-window-origin isa exception-class          ; the new exception
    superset window-larger-than-screen]           ; it's position in the hierarchy]
```

When *wrong-window-origin* is raised, the following information is reported.

```
*** Window W : offset + length exceed screen length.
*** Exception signaled while sending message display-window to *current-terminal*.
1 : Abort
2 : Inspect or modify W and retry operation.
3 : Display only the visible part of W, starting from left corner.
4 : Wrap W around the screen.
....
```

[5]a class mainly designed to be a place where common pieces of behavior are grouped together, rather than to be instantiated.

Other important consequences of the hierarchical organization lie in the ability to signal and handle multiple exceptions; we develop these points in the following paragraphs.

1.3. Signaling an exceptional event.

As far as exceptions are implemented as classes, it is straightforward to compute an occurrence of an exceptional event by an **instantiation of the related exception**. From a syntactic point of view, signaling simply consists in sending the message **signal** to an exception. This method is nothing but a redefinition of the method **new** on the class *exception-class*. It first creates an instance of the signaled exception and then searches a suitable handler.

What are the interesting points and, eventually, the advantages of this protocol?

First of all, instantiation is a basic operation of an OOL, is efficient and free to implement. Furthermore, as for any instantiation, all the slots defined on the exception can be set at signaling time (if no values are provided, the default ones are used). Here is an example of raising the exception *window-larger-than-screen*, where one slot is explicitly set.

```
; body of the method "display-window" defined on "terminal",
; oself == current terminal, w == a parameter of type "window".
if [ [w length] > [oself length] ]
        then [window-larger-than-screen signal  window-to-be-displayed: w]
```

Besides, the two slots *xobj* and *xprop* defined on the exception lattice root (cf. § "Properties defined on exceptions") are set by the system in order to store the signaling context. Here is (a part of) what can be learned within a handler, by inspecting the instance of the signaled exception.

```
; inspection of the instance of the exception
is-a: (<window-larger-than-screen>)
name: xself
xobj: <*current-terminal*>
xprop: <display-window.terminal>
xargs: <unknown>
xhandler: "*<protect-handler> for (<window-larger-than-screen>)*"
window-to-be-displayed: <w>
...
```

• Parameterization.

In *Ada* or *PL/I*, exceptions cannot have parameters; thus, information cannot be conveyed from the signaler to the handlers; these programs can only communicate through global variables, leading to some well-known programming problems. In many other languages such as *Clu*, *Mesa* or in Knudsen's proposal [Knudsen 87], solutions for associating parameters with exceptions are provided; parameter names (and types) are declared either in each handler heading (*Clu*) or while declaring the exception (*Mesa*). A handler can only be invoked if the signaling arguments match with the parameter types. Subsequently, in order to allow handlers to trap all exceptions exceptions, whatever arguments are passed while they are signaled, some kind of pattern-matching star conventions must be designed. Such mechanisms are powerful, but they might be complex to implement.

Our model provides a solution to parameterization which is simple both to implement and to describe. All the handlers receive a unique and implicit argument which is the previously described

instance of the current exception[6]. Then, no complex heading is needed to define handlers. This instance holds all the possible pieces of informations about the exception.

• Semantics of signaling.

It is generally agreed that a signaler knows the seriousness of a situation, whereas a caller can handle an exception with full knowledge of the situation. Subsequently, a signaler is responsible for choosing between signaling either a fatal or a continuable exception while a handler is responsible for resuming, exiting or propagating the exception.

Within standard exception handling systems, a set of primitives is generally provided to support the various cases; e.g., in Goodenough's proposal, signaling with *escape* states that termination is mandatory, signaling with *notify* forces resumption and signaling with *signal* let the handler responsible for the decision. By the way, a set of primitives is also needed to achieve resumption and termination.

We do not need to provide such a set of primitives since the set of actions that can be performed by handlers only depends on the signaled exception. For example, as we shall see (cf. § "handling exceptions"), it is by construction impossible to resume from an *error*. Thus, *signal* is the only signaling primitive.

• Signaling multiple exceptions.

Our system allows users to signal multiple exceptions, i.e. exceptions that are some nodes in our hierarchy. This property, although not fundamental may be useful is some situations. For example, signaling a *warning* or an *arithmetic-exception* may be useful either to report very general situations or to avoid the creation of a specific exception-class. This may also be useful to hide a very low-level exception : indeed, a handler is able to catch the low-level exception and to propagate an upper type of it.

1.4. Handling exceptions.

Handlers are responsible for resuming, exiting or propagating the exception. Object oriented handlers own the following advantages over more classical ones: handlers of various conceptual levels can be distinguished even if all are first-class entities; they are able to handle multiple exceptions; finally, they own simple protocols to handle exceptions.

• Various kind of handlers

Responses to exceptional events may belong to different conceptual levels. These differences can be expressed thanks to various handler attachment capabilities. We allow users to attach handlers to exceptions, to object classes and to any expression of our language.

The most general (default-)handlers are independent of any execution context as well as of the state of the object data base. In our system, these handlers are methods defined on exception themselves (cf. the method *handles-default* defined on the exception lattice root, § 1.2). It is then possible to determine statically by inspecting the object representing an exception which default-actions may be performed in case it is raised and not handled. Specific exceptions inherit from the default-handlers defined on upper ones (cf. § 1.2 the "abort" proposition defined on *error*); general default-handlers can

[6]It is important to notice that in order to determine whether a handler is to be invoked, the system tests whether its parameter type is an upper type of the signaled exception.

be, entirely or partly (cf. the method ''describe-exception'') redefined. In our system, all properties, notably the methods are also first class entities (each method owns several slots that contain its definition domain, its range, its parameter list, and of course its body); thus, default-handlers themselves can be entirely inspected.

Handlers attached to other classes are also classical methods, they allow to ensure that an exception cannot be raised outside of a method invoked by sending a message toward an object of that class. This issue being very specific to OOP, we do not say more about it.

More specific handlers with dynamic extent, attached to instructions, are not attached to exceptions but dynamically stack-allocated. However, they are instances of a specific class and can also be inspected. To do that, we define a primitive that dynamically returns an ordered list of handlers that would be invoked if its exception argument were to be raised. One could argue that a slot should be defined on *exceptional-event* to contain, at any time, all the handlers related to the exception; this would be very complex and time-consuming because of the ability for a handler to trap multiple exceptions.

• Protocol for handling.

Inheritance and message sending rules make all handling operations generic, i.e we (exception handling system implementors) do not have to perform any test to ensure that a programmer will not attempt to resume from a fatal exception and vice versa. Indeed, let us recall that all handlers have one parameter which is the instance of the currently signaled exception; then, either resumption or termination can be achieved within handlers by sending the messages **resume** or **exit** to the instance of the exception. The method *resume* is defined on *proceedable-event* whereas *exit* is defined on *fatal-event* (cf. fig. 2); thus, sending the message *resume* to an *error* will raise the exception *unknown-selector*.

Let us say a little more about resumption. Resumption sometimes needs a consensus between the signaler and the handler; notably in the cases where the handler is responsible for saying what to do, but where the operations allowing to restart computation must be performed by the signaler in its environment. For example, when a handler proposes to display only the visible part of a window, the routine that signaled the exception must have foreseen this possibility since it will have to manage the operation. In the *Flavor* object oriented exception handling system, a slot is designed for specifying, at signaling time, which solutions, among all the possible ones created to resume from the current exception, are available in this particular case. This is an original and very useful issue since it allows users to specify protocols for resumption that are controlled by signalers. Handlers may also judge in full knowledge of the facts.

• Handling multiple exceptions.

Multiple exceptions can be propagated, and handled too. This means that all the handlers are aware of the exception hierarchy, defining a handler for an exception amounts to defining a handler for all exceptions that are subclasses of it. Thus, any (may be unexpected) exception which is a sub-exception of the exception for which the handler has been designed, can be caught by defining a sole handler. For example, any handlers for *arithmetic-exception* catches just as well *overflow*, *zero-divide* or *arithmetic-exception* itself.

2. How far exception handling improves reusability in object-oriented languages.

2.1. Why do object-oriented languages improve reusability and extendibility.

An interesting analysis on that subject can be found in [Meyer 87]; here are some key ideas and pieces of examples.

- Abstract data types do exist and new ones can be defined. Instantiating an abstract data type is a first kind of reusability. Once a new object is created, and particular values given to its slots, all the operations that are properties of its type can be applied to those values. Inheritance allows to propagate on a new created type all the knowledge of its ancestors in the hierarchy.
- Concrete actions can be overloaded on subclasses, A frequently quoted rule for writing reusable programs is to separate what is general and what is peculiar, the abstract from the concrete. To write reusable programs means defining abstract (or generic) versions of operations on abstract classes; these operations are generic since they only make use of lower level abstract operations (let us call them sub-operations) instead of directly performing concrete actions or side-effects. Sub-operations can be defined (or redefined, when default-sub-operations have been provided) on subclasses that implement real data types.

- The method *handles-default*, defined on *exceptional-event*, with its four compound operations (e.g. *describe-exception*) (cf. fig. 3) is a first example of an abstract operation. Its sub-operations have got general definitions on the same abstract class (e.g. *describe-exception* on *exceptional-event* reports a very general error message including the exception name). This ensures that *handles-default* will never raise the exception *unknown-selector*. Thereafter, any sub-operation can be redefined on a subclass (e.g. *describe-exception* on *window-larger-than-screen*).
- Here is another example slightly different: the class *sequence* is defined with two basic sub-classes: *list* and *string*. Here is a classical version of a generic method "*append*" defined on *sequence*.

```
; s1 and s2 are sequences to be appended.
if [s1 empty-p]
    then s2
    else [[s1 first] put-first [[s1 all-but-first] append s2]
```

  Defining the sub-operations: "*empty-p*", "*first*" and the like on *sequence* would not make sense, unless to signal the error *subclass-responsibility* [Goldberg 83]; they have to be defined on *sequence*'s subclasses. Afterwards, *sequence* can be refined with new data-types. For example, sending the message *append* to a vector would result in signaling the *unknown-selector* exception. However, it is straightforward to extend the *append* operation to vectors by defining the class *vector* as a subclass of *sequence* and by providing on it the required operations.

- Finally, message sending extends overloading by introducing within each communication an indirection via the message receiver which allows each new definition of a sub-operation to be taken into account even when abstract operations have been compiled.

2.2. Why does exception handling improve reusability.

Exception handling improves reusability as far as it is not only used to write fault-tolerant programs but is also used as a way to extend the definition domain (by handling domain exceptions) as well as the functionalities (by handling range exceptions) of an existing operation.

Seeing that "subclassing" is the most usual way to extend the definition domain of an operation (cf. the above *sequence* example), the question now is: why does the exception handling system offer extra functionalities? The answer is: because redefining sub-operations on subclasses is not always possible nor always what is wanted.

• **Subclassing is not always what is wanted.**

With subclassing, operations can be extended to, or specialized for, new set of objects. Furthermore, since methods are global objects, the scope and the extent of these extensions are indefinite. But what about extending the definition domain of an existing method only for a particular execution? Such an issue can only be completed by attaching a handler to the instruction performing this particular call. For example, a programmer might desire to trap an occurrence of the exception *wrong-window-origin* and to entail the *wrap-around* resumption, without definitely choosing such a solution for further calls of the method *display-window*.

Similarly, attaching handlers to instructions allows to handle differently the exceptions raised by sending the same message to various objects of the same class, or even to the same object. For example, various resumption solutions can be specified in handlers attached to different invocation of the method *display-window*.

Besides, exception handling also provides new subclassing capabilities as far as handlers can be attached to classes as in *Smalltalk, Lore*. Indeed, instead of redefining an operation O on a subclass SC, a handler for an exception E raised by O can be defined on SC. Afterwards, every time E is raised by O after a message sending towards an instance of SC, the handler will be invoked. The same result would be obtained by redefining O on SC and by replacing the instruction *signal* by the handler's body. This second solution is obviously more complex and does not make use of the upper definition.

In our window example, it is thus possible to define a subclass of *terminal*, named *wrap-around-terminal* to which a handler for *wrong-window-origin* is associated. Afterwards, as far as the object **current-terminal** belongs to *wrap-around-terminal*, the "wrap-around" resumption is performed each time the exception *wrong-window-origin* is raised, without it being necessary to attach the handler to each call to *display-window*.

• **Subclassing is not always possible.**

Subclassing is not possible when the definition domain of a new operation cannot be created in the language, or when the objects that constitute this domain already exists in the language and subsequently, cannot be created again as instances of a new class.

Infinite sets of predefined objects usually confront programmers with such issues. For example, every OOL implements the type **integer**, each integer being encoded to belong to this class. It is generally impossible[7] to define the set "{1 2 3}" or the set of positive integers, and furthermore to define methods on them. Then, if a method is to be defined, the domain of which is *positive-integer*, it has to

[7]This is not true of every OOLs, but such distinctions are beyond the scope of this paper.

be defined on integers and it is useful to be able to signal an exception when the related message is sent to a negative number.

Besides, assume that a method "M" is defined on the class *sequence* to compute the successive division upon all the elements (assuming these are integers) of a particular sequence; if 0 belongs to the sequence, *zero-divide* will be raised. If it is wished for a particular call to "M" that division by zero return the greatest integer in the machine, the solution is to associate a handler to that invocation of "M". Indeed, it is not possible to redefine *divide* on the integer's subset {0}. As can be seen, even if it were possible, it is not what it is wanted because this extension of the definition domain would then be permanent.

• **Other issues.**

A final issue is that exception handling allows users to write fault-tolerant and reliable programs that are reusable. Any method owns an interface specifying its expected arguments, the type of its result and the exceptions it may signal. When sub-operations used by a generic method are likely to be redefined by users on new data types, it may be important that methods be able to trap all the unexpected low-level exceptions raised by these sub-routines, and be able to propagate the higher level ones declared in its interface. Here again, *handles-default* is a good example. Since it is the most general handler, it is mandatory that it not raise exceptions, or more precisely that it handle all those raised by its embedded sub-operations; otherwise, the system could never be reset to a coherent state.

Conclusion

In this paper, we mainly explain how giving exceptions the status of data types, and to occurrences of exceptional events the status of elements of those types, fits well to implement an efficient exception handling system. We show that object oriented languages are especially well-suited to satisfy this requirement. Exception handling is related to software quality for it allows writing fault-tolerant and reliable programs.

Object oriented languages provide a software decomposition technique well-suited for writing reusable code. We have explained how our exception handling system is itself extensible and reusable, and how it further improves reusability.

Implementing exceptions as data types have other applications in fields such as information systems (in order to store in data-bases, some exceptional data, for example with slot values breaking constraints [Bordiga 86]), or such as debugging tools (in order to provide user-friendly responses after users program failures [Lieberman 87]).

The work described here is experimental. Our system is implemented within the *Lore* object-oriented language and is used, mostly in simulation and AI applications.

References.

[Bidoit 84] M. Bidoit, M.-C. Gaudel, G. Guiho : Towards a systematic and safe programming of exception handling in Ada Proc. of the Third Joint Ada Europe/AdaTEC Conference, Brussels, June 1984, The Ada Companion Series, Cambridge University Press, pp.141-152.

[Bidoit 85] M. Bidoit et al. : Exception Handling: Formal Specification and Systematic Program Construction I.E.E.E. Transactions on Software Engineering, Vol. SE-11, Number 3, March 1985, pp.242-252.

[Booch 86] G.Booch : Object Oriented Development. IEEE Transactions On Software Engineering, Vol SE-12,

No 2, February 1986.
[Borgida 86] A.Borgida : Exceptions in Object-Oriented Languages. ACM Sigplan Notices, Vol. 21, No. 10, pp. 107-119, October 1986.
[Caseau 86] C.Benoit, Y.Caseau, C.Pherivong : Knowledge Representation and Communication Mechanism in Lore. Proc. of ECAI'86, Brighton, July 1986.
[Caseau 87] Y.Caseau : Etude et Réalisation d'un langage objet : LORE. *Thèse de l'université Paris-Sud*, Orsay, France, Novembre 1987.
[Christian 82] F.Christian : Exception Handling and Software Fault Tolerance, IEEE Transactions on Computers, Vol. C-31, No. 6, pp. 531-540, June 1982.
[Cox 83] B.J.Cox : The Message/Object Programming Model A small change at a deep conceptual level. In Proceedings of SoftFair, (IEEE Order No 83CH1919-0), July 83.
[Dahl 70] O.Dahl, B.Myhrhaug, K.Nygaard : SIMULA-67 Common Base Language. SIMULA Information, S-22 Norwegian Computing Center, Oslo, Norway, October 1970.
[Dony 88] C.Dony : An Exception Handling System for an Object-Oriented Language. To appear in proc. of European Conference on Object Oriented Programming, Oslo, Norway, Aug. 1988.
[Goldberg, kay 72] A. Goldberg, A. Kay : SMALLTALK-72 Instruction Manual. Memo SSL 76-6, Xerox Palo Alto Research Center, Palo Alto (CA) mars 1976.
[Goldberg, Robson 83] A. Goldberg, D. Robson : SMALLTALK 80, the language and its implementation. Addison Wesley 1983.
[Goodenough 75] J.B.Goodenough : Exception Handling: Issues and a Proposed Notation. Communication of the ACM, Vol. 18, No. 12, pp. 683-696, December 1975.
[Ichbiah 79] J.Ichbiah & al : Preliminary ADA Reference Manual. Rationale for the Design of the ADA Programming Language. Sigplan Notices Vol. 14, No. 6, June 1979.
[Knudsen 87] J.L.Knudsen : Better Exception Handling in Block Structured Systems. IEEE Software, pp 40-49, May 1987.
[Levin 77] R.Levin : Program structures for exceptinal condition handling. Ph.D. dissertation, Dept. Comput. Sci., Carnegie-Mellon University Pittsburg, June 1977.
[Lieberman 86] H.Lieberman : Delegation and Inheritance, Two Mechanisms for Sharing Knowledge in Object-Oriented Systems. 3rd AFCET Workshop on Object-Oriented Programming, J.Bezivin and P.Cointe (ed.), Globule+Bigre, No 48, pp 79-89, Paris, Janvier 1986.
[Lieberman 87] H.Lieberman : Reversible Object-Oriented Interpreters. in Proceedings of European Conference on Object-Oriented Programming (ECOOP'87), special issue if BIGRE No 54, pp 13-22, June 1987, Paris.
[Liskov 79] B.Liskov, A.Snyder : Exception Handling in CLU. IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, pp. 546-558, Nov 1979.
[Meyer 87] Reusability: The Case for Object-Oriented Design. IEEE Software, pp. 51-64, Mars 1987.
[Moon 83] D. Moon, D. Weinreb : Signalling and Handling Conditions, LISP Machine Manual, Fourth Edition. MIT Artificial Intelligence Lab., Cambridge, Massachussets, (July 1981).
[Nixon 83] B.A.Nixon : A Taxis Compiler. Tech. Report 33, Comp. Sci. Dept., Univ. of Toronto, April 83.

VALIDATION BY PRE-REVIEWING AND JUSTIFICATION

Ilkka Tervonen
Institute of Information Processing Science
University of Oulu
Linnanmaa, SF-90570, Oulu
Finland

ABSTRACT

The paper deals with software validation in a three level development process which is tailored to embedded software. The ESD approach, used as the background for the thesis (based on three description levels, E=Embedded, S=Specification, D=Design), also formulates the main principles for the validation of embedded software. According to these principles, each description level has its own quality features which focus interest on specific characteristics and define the reasons for the selection of one description among several alternatives. The quality features are total effectiveness at the E level, functional impact at the S level and total efficiency at the D level.

The paper decomposes the validation process. The specific characteristics of validation and validation techniques are considered in the context of the ESD approach and a review method based on pre-reviewing of the descriptions and justification of the design decisions is presented. The pre-reviewing and justification principles are illustrated in a prototype.

BIOGRAPHICAL SKETCH

Ilkka Tervonen is currently an Associate Professor (acting) of Information Processing Science at the University of Oulu. He has been involved in research projects including SAMPO (to produce a Logo-like programming language), CONSE (to produce a conceptual model for embedded software and its production), and OSAAJA (to produce the prototype described in this paper).

He received the degree of Cand. Phil. from the University of Oulu, and his research interests include conceptual modelling of SE, software testing, development environments, metrics, and AI-SE.

VALIDATION BY PRE-REVIEWING AND JUSTIFICATION

Ilkka Tervonen
Institute of Information Processing Science
University of Oulu
Linnanmaa, SF-90570, Oulu
Finland

INTRODUCTION

This paper continues the research work on conceptual modelling in the Software Engineering area begun at the University of Oulu in 1985 under the CONSE project[1], in which Iivari et al. (1986a, 1986b) introduced a conceptual model for embedded software and its production. The present layered approach for the testing and particularly the validation of embedded software has its roots in that conceptual model.

Using this conceptual model this paper will further explore the validation of embedded software. The main idea in the CONSE approach was to shift the "complexity barrier" to the product level and in this way to achieve the cognizance of the product necessary to define and develop the software. The same idea is also visible in other papers concerning embedded software (cf. Zave 1982, Valmari 1987), although our presentation goes further and depicts a specific product model as a starting point of its development. This approach does not address itself to the behavioural perspective or to the dynamic aspects of software, although one must admit the importance of modelling the temporal behaviour and timing estimates of software (cf. Wallace et al. 1987), especially in the context of embedded systems.

The "deterministic" product environment, according to the CONSE research, is the most fundamental difference between completely embedded software and traditional information systems and related software, which function in an organizational environment which is essentially non-deterministic and fuzzy due to the intentional human action or behaviour involved. Owing to the real-time characteristic of embedded software (requirement of the embedded system), which causes "unpredictable program behaviour" (cf. Brinch-Hansen 1973), the property of being "deterministic" is not relevant at the software design and implementation levels.

The problem of testing embedded systems and software has been recognized in the SE area. Glass (1980) uses the phrase "the lost world" and considers the gap between methodologists and practitioners. Although Glass paints quite a pessimistic picture of the testing of embedded

[1] CONSE = CONceptual modelling of Software production and Engineering

software, new design methods and principles give us some hope of achieving this. These methods provide different perspectives on the development of the software. One example is RT-SA/SD (Real Time Structured Analysis/ Structured Design), which offers functional, behavioural and information modelling perspectives (Ward and Mellor 1985). It is to a great extent consistent with the ESD approach and models, and provides a practicable way of understanding the principles and possible applications of ESD models, which due to their metamodel-like nature are not directly usable in practice. The RT-SA/SD method is also employed as a link between the ESD approach and the present prototype, which supports a type of validation technique.

This paper first describes the three levels of the ESD approach and validation as a part of the ESD model. The next part discusses the validation techniques and then validation by pre-reviewing and justification is presented. The role of rationale in the method is discussed after that, and finally, these ideas are illustrated in a prototype.

VALIDATION AS A PART OF THE ESD APPROACH

The ESD approach consists of three models; the ESD model for Embedded Software, the ESD model for the Development of Embedded Software, and the ESD model for the Quality of Embedded Software. Since we have introduced the ESD models for an Embedded Software and for its Development in detail in other papers (Iivari et al. 1986a, 1986b, Iivari 1987), we next briefly represent the three levels which form the basis for the ESD approach.

1. Embedded (E) level: The model at the E level describes the embedded software in terms of the functions of the product controlled or performed by the software and in terms of the interface between the software and the surrounding product.

2. Software specification (S) level: The model at the S level describes the primary information processing at the conceptual/infological (logical) level performed by the software in terms of semantically well-defined information and information process types and the interaction between the software and its environment.

3. Software design (D) level: The model at the D level describes the primary data processing or computing at the technical/datalogical (physical) level corresponding to the specifications and the secondary data processing or computing necessitated by various internal control and supporting functions in the software.

If we now outline the characteristics of validation in the ESD approach, we obtain the tree presented in Figure 1. A specific characteristic of the development of embedded software is the classification into host system development

and target system development. Testing follows the same classification and in this paper we discuss about host system testing, since the testing environments are more advanced in this branch of testing.

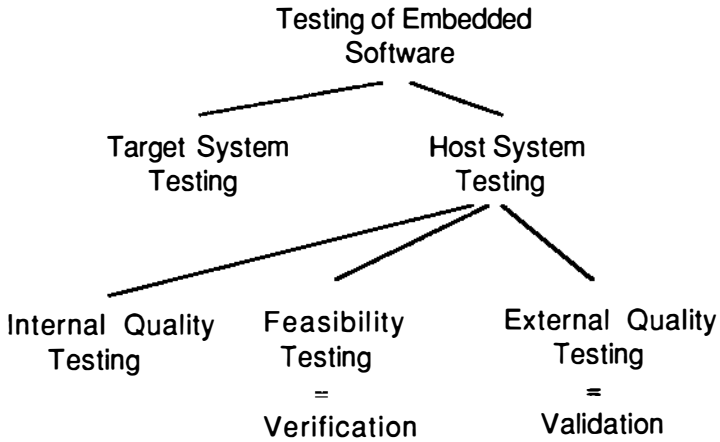


Figure 1 Outline of testing types

According to the ESD model the basic testing types are, internal quality testing, feasibility testing and external quality testing. These testing types have special characteristics such that internal quality testing is based on the grammar (set of sentences and code of reasoning) evaluation, feasibility testing considers the satisfiability of the descriptions to the upper level, and external quality testing assesses the specific quality features at each description levels.

We shall go on in this chapter to consider the testing types, the role of requirements in it and the consequences of learning dynamics for verification and validation.

Internal quality testing

Internal quality testing concerns the form and syntax of the descriptions. As its name suggests the internal testing tests the descriptions with respect to the grammar of the descriptions and the two specific characteristics are consistency and completeness.

A formal definition, presented by Turski and Maibaum (1987), gives the most exact description of these characteristics. The consistency is defined "a set of sentences G is consistent if and only if there is no sentence A such that both A and ¬A are derivable from G", and completeness "a set of sentences G is said to be complete if for any sentence A either A ϵ G or ¬A ϵ G".

Depending on the level of formality of the grammar the definitions of consistency and completeness vary correspondingly. Boehm (1984), for example, has assessed

completeness of requirements and design specifications to the extent that all of its parts are present and each part is fully developed, i.e. no TBDs (to be determined), no noexistent references, no missing specification items.

Feasibility testing, or verification

Feasibility testing in the ESD approach is very closely confined to verification, which means that all upper level definitions are traceable to lower level ones and the lower level descriptions satisfy the upper level ones.

A simplified description of feasibility testing is presented in Figure 2.

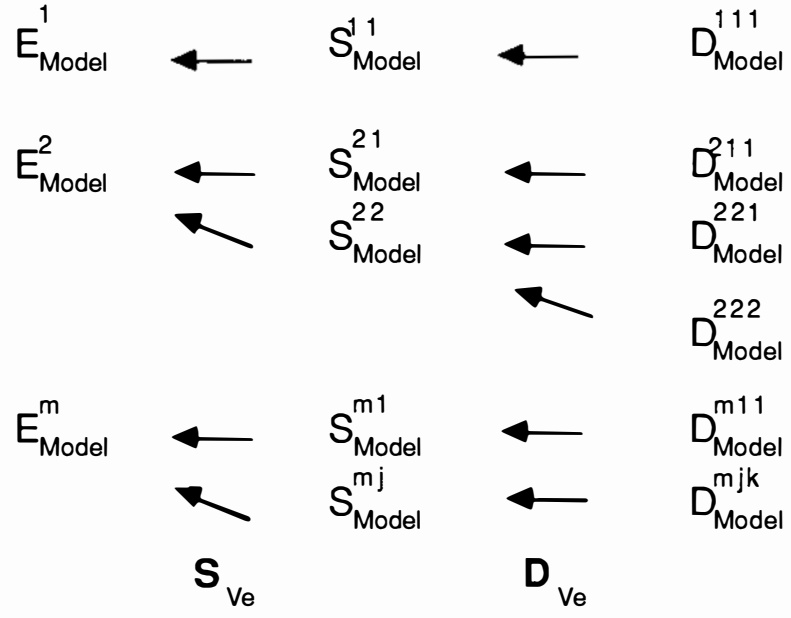


Figure 2 Verification

This simplified version does not mention the learning dynamics involved in development, which will be introduced later, in Figure 4. The formula for simplified verification is as follows:

$$S^{mj}_{Ve} [\, S^{mj}_{Model} , E^{m}_{Model} \,]$$

External quality testing, or validation

External quality testing inherits its name from external referent (i.e. customer needs, external quality requirements etc.), to which the software is assessed. External quality testing or validation is the most essential testing activity from the ESD model point of view. The main idea in the validation of descriptions is to select the appropriate alternative at each level, based on external quality requirements. The validation itself is then classified into requirements validation and models

validation. These principles, excluding learning dynamics, are presented in Figure 3.

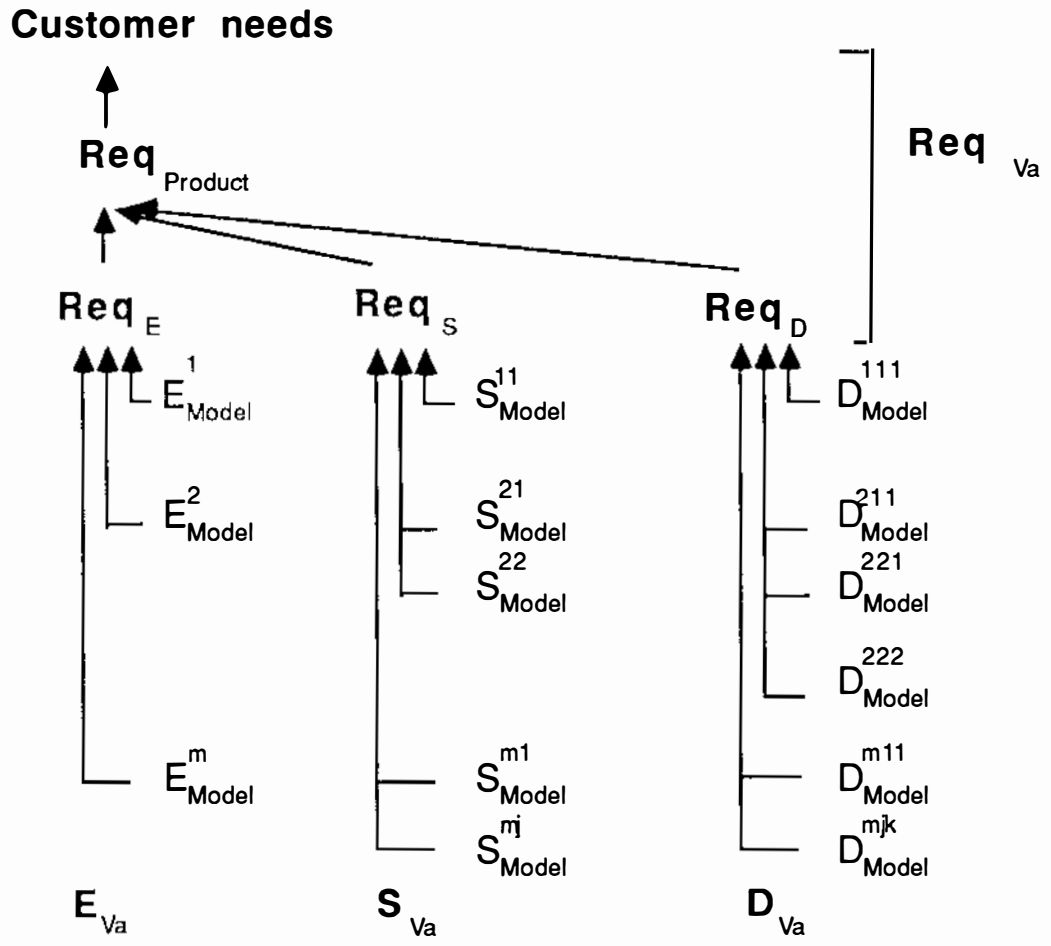


Figure 3 Validation

The requirements validation of embedded software takes place in two parts, the product requirements are first validated and then decomposed into their different description levels. It is important to note the distinction between customer needs and product requirements, which is to a great extent similar to that between user needs and user requirements in the area of conventional software. From the software point of view the product model allows the customer easier to understand the behaviour of the whole product than only that of the software part. The point in this emphasis on product requirements is that they are used as a first definition and starting point for software development.

According to the specific characteristics of the ESD approach at different levels of description, the selection of an alternative is based on different quality features such as total effectiveness at the E level (the total impact of the software component on the quality of the product and its costs), functional impact at the S level

(the functional impact of the functional accuracy, usability, adaptability and maintainability on the corresponding quality factors of the product) and total efficiency at the D level (efficiency of the software implementation).

Validation of the descriptions at the different levels is based on testing against the validated requirements. Due to the non-linear character of the ESD approach, validation also includes technical and economic evaluation of the descriptions (at lower levels) which is included in feasibility testing and verification in some approaches.

The formula for the validation process can be represented at the E level for example by

$$E^{m}_{Va} [E^{m}_{Model} , Req_{E}]$$

The formulae at the S and D levels are equivalent. This formula does not include the learning dynamics, which is a major principle in the ESD model for the Development of Embedded Software.

Synthesis of verification and validation

If we extend the formula of validation to include the learning dynamics as well, we obtain the next form

$$E^{m}_{Va} (n_{im}) [E^{m}_{Model} (n_{im}) , Req_{E} (n_{ie}) , k^{m}] ,$$

where

(n_{im}) = (n_{im}) th version of E^{m}_{Model}

(n_{ie}) = (n_{ie}) th version of Req_{E}

k^{m} = cumulative knowledge concerning Model m

Analogous formulae can also be presented for validation at each description level. That for verification at the S description level is

$$S^{mj}_{Ve} (n_{imj}) [S^{mj}_{Model} (n_{imj}) , E^{m}_{Model} (n_{im}) , k^{m}]$$

The cumulative knowledge k^{m} , concerning Model m justifies the selection of the description, say, $S^{mj}_{Model} (n_{imj})$. As we notice from the formulae, this knowledge is gathered during both verification and validation.

As a final representation, we depict in Figure 4 a combination of verification and validation with the learning dynamics.

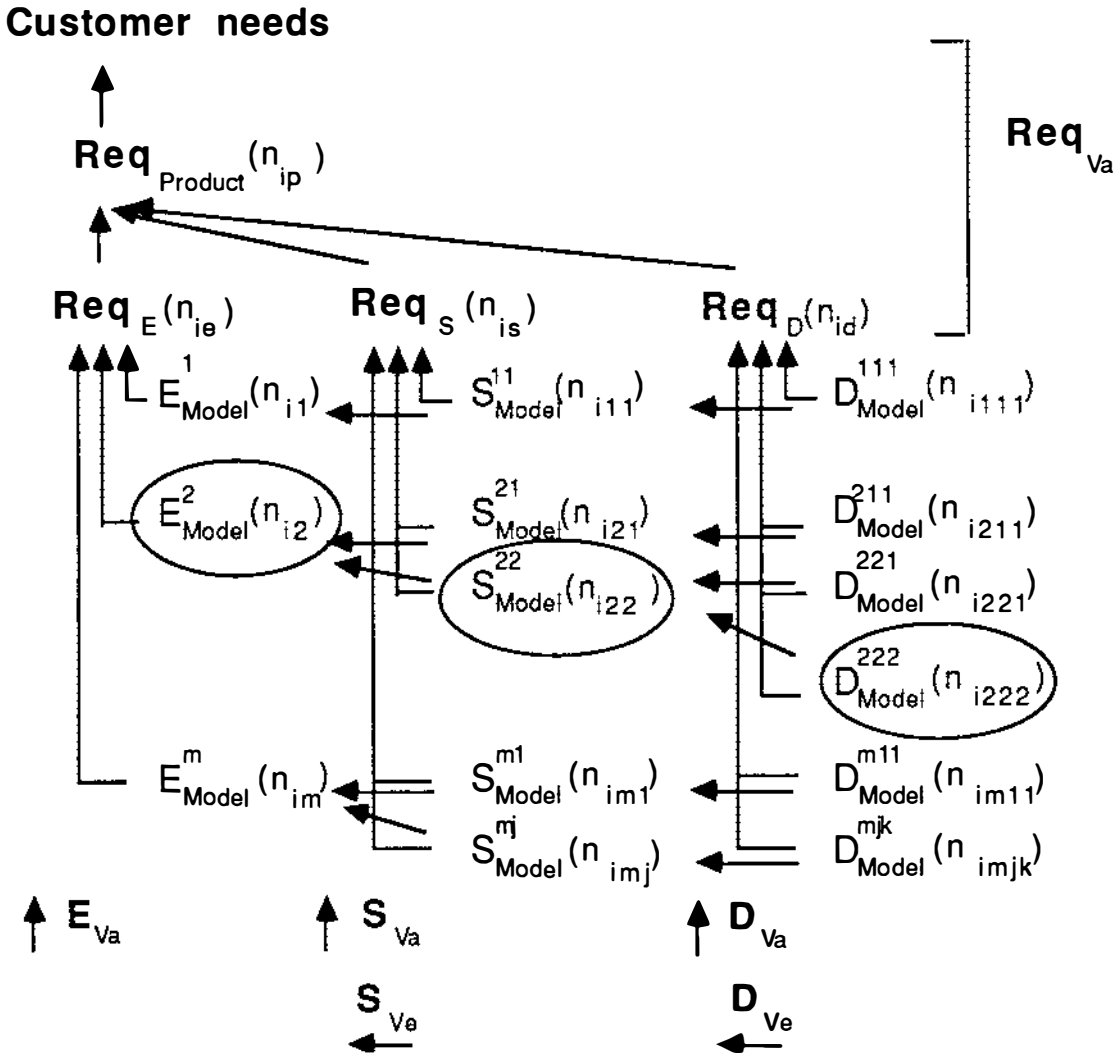


Figure 4 Verification and validation

The alternatives selected at each level are marked with circles. Non-linear development means that although we have finally selected version (n_{i22}) of the S^{22}_{Model} at the S level, for example, we have also assessed some technical and economic characteristics of other models at the D level (e.g. the D^{211}_{Model} in relation to S^{21}_{Model}).

This definition and the formulae are important in more senses than one, since they demonstrate (1) the difference between verification and validation, (2) the fusion of verification and validation in practice, (3) the learning dynamics in the case of verification and validation, (4) the main ideas of quality features as a background for decisions concerning the selection of alternative descriptions (discussed in more detail later in this paper).

After all the main purpose of our validation approach is; "to decompose the validation process into different

description levels". This means that at the E level validation is equivalent to the assessment of total effectiveness of the product, at the S level to the assessment of functional impact upon the product quality and at the D level to the assessment of total efficiency.

VALIDATION TECHNIQUES

In Figure 1 we described the upper level structure of the present validation concept. At the next, more detailed level we consider validation techniques using classification depicted in Figure 5.

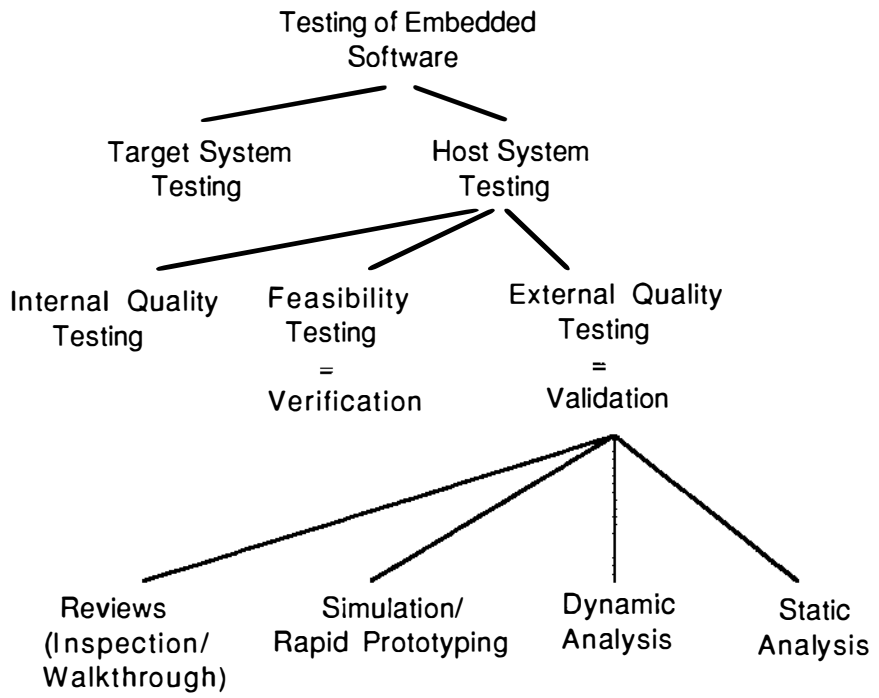


Figure 5 Validation techniques

Some of the techniques may be common to all testing types or some of them, but we consider them here only from the validation viewpoint. We define these techniques briefly before we assess their relation to the specific levels of description.

Reviews: The principle of reviews is under an uninterrupted meeting, (i.e. team consists of three to five people, duration one to two hours), to check descriptions/ code. Inspections are based on checklists and walkthroughs on mental execution of test cases.

Simulation/ Rapid Prototyping: In the present taxonomy, where simulation is excluded from dynamic analysis and considered as own type, this class is focused on the analysis of behaviour of the product or embedded software at implementation-independent level.

Dynamic analysis: Dynamic analysis is technique used to derive meaningful information about execution behaviour of the software. In our taxonomy the analysis of behaviour in advanced testing environments is included in simulation/rapid prototyping class and dynamic analysis techniques are for example assertion checking, coverage analysis, regression testing and tracing/debugging.

Static analysis: Static analysis is technique which supports the analysis of syntax and semantics of the software (e.g. auditing, completeness and consistency checking, complexity measurement, scanning, structure checking, reference analysis). The essential functions of static analysis are to check whether descriptions are consistent, non-contradictory or unambiguous and to check the characteristics of internal quality in our taxonomy.

The degree of relevance is presented in Figure 6 using a three-level scale; high, moderate, and low.

	Reviews (Inspection/ Walkthrough)	Simulation/ Rapid Prototyping	Dynamic Analysis	Static Analysis
E level	Moderate	High	Low	Low
S level	High	High	High	Moderate
D level	High	Moderate	High	High

Figure 6 Degree of relevance

Reviews: It is possible to achieve a more accurate assessment of descriptions at the S and D levels than at the E level because the checklists at these levels are better. Due to the importance of correcting any errors as early as possible, the reviews also have some value at the E level.

Simulation/Rapid Prototyping: Prototyping is the most important method available for understanding the main functions of the product. In the case of embedded systems this understanding is necessary to define the requirements of the embedded software. At the S level, for example, executable specifications are necessary in the same way in order to define the correct and relevant specifications. Executable specifications shift prototyping also to the D level, since timing analysis is to a great extent analysis of the D level characteristics.

Dynamic Analysis: Due to the executing nature of dynamic analysis, the D level is naturally the main description level for this kind of validation. Because the specification languages used nowadays are more formal and there is some supportive environment behind them, dynamic analysis is also possible at the S level. In our taxonomy we shifted advanced environments, which support behavioural analysis, to the simulation/rapid prototyping class.

Static Analysis: Validation by static analysis is traditionally one of the D level validation techniques, but the attempts made to use more formal languages in the definitions of specifications and requirements also allows us to use static analysis at the S and E levels to a certain extent.

VALIDATION BY PRE-REVIEWING AND JUSTIFICATION

We focus in the latter part of the paper on review type of validation. The inspection and walkthrough principles of Fagan (1976) have been the most popular informal validation techniques in recent years, while Gilb's Design by Objectives principle (Gilb and Krzanik 1987) and the ideas of Hausen (1987) and the REQUEST project (Kitchenham 1987) have developed this branch of testing techniques still further. These principles also serve as examples of the operationalization of quality factors. Pre-reviewing and justification in Figure 7 follow this operationalization principle and are instances of reviewing.

The one branch of reviewing is based on the idea that design descriptions (e.g. PEM, SEM or COM descriptions at the physical level according to Yourdon 1984) are pre-reviewed using a "quality checklist" adopted from the SE literature or from experiences of the company concerning quality metrics, method-independent design principles or method-dependent design principles. The assessment carried out in pre-reviewing is analogous to conventional reviewing and records the rationale of the selection. The difference in relation to reviewing is its connection with the design process and the fact that the assessment is produced by an individual designer and not a team.

Another branch of reviewing, justification, is connected with design decisions, which can be (1) aggregate decisions concerning selection among alternative descriptions (cf. validation in the ESD approach) or (2) individual design decisions concerning task decomposition in the SEM model, for example (cf. Yourdon 1984). Justification is a solution to meet future quality demands, where software designers must be able to make better and more informed decisions and it is based on method-independent design principles or method-dependent design principles.

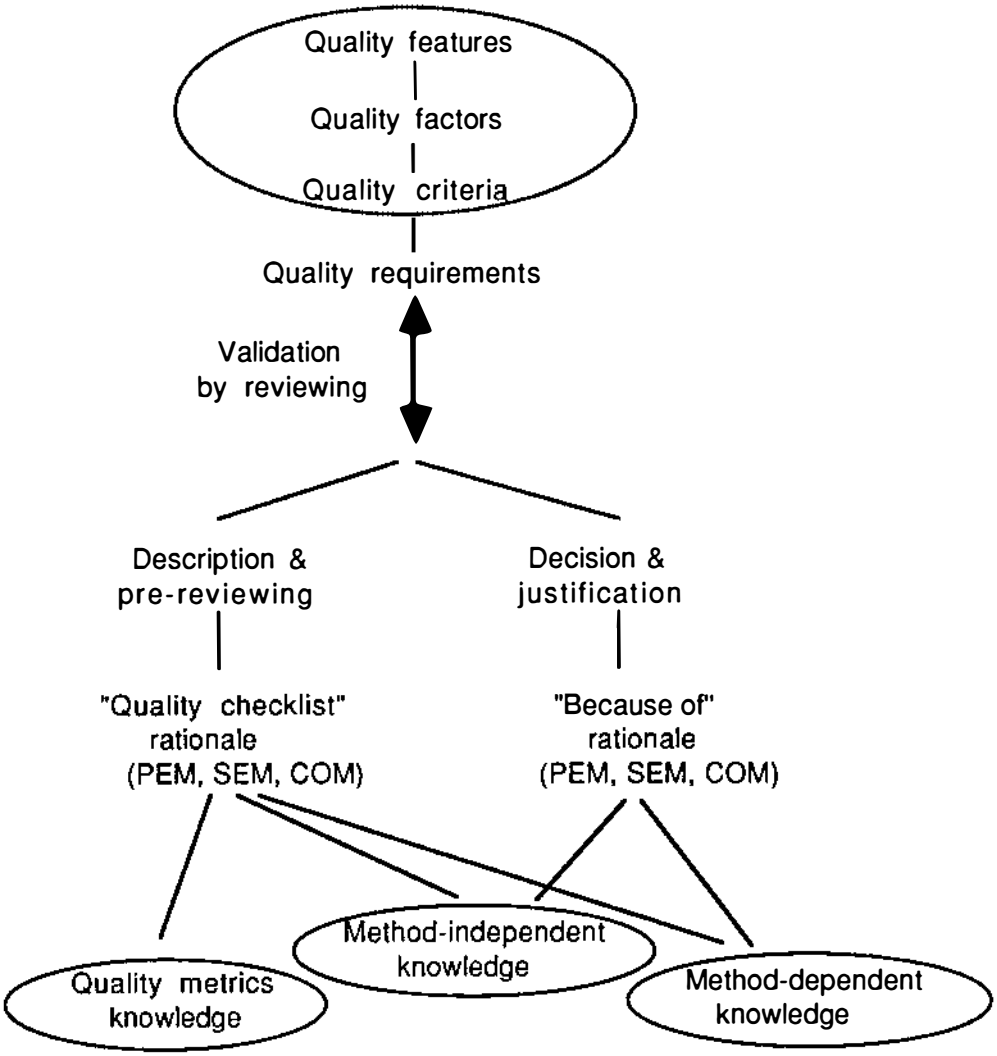


Figure 7 Two branches of reviewing

The potential quality factors, which refine the quality features at each description level, are depicted in Table 2. A more detailed definition of the factors is presented in Tervonen and Alanko (1988). The letters P and S refer to the product and software viewpoints.

Embedded level	**Specification level**	**Design level**
total effectiveness	functional impact	total efficiency
functional accuracy (P)	functional accuracy (P/S)	performance (S)
area of operation (P)	usability (P/S)	adaptability (S)
robustness (P)	adaptability (P/S)	maintainability (S)
duplicability (P)	maintainability (P/S)	
maintainability (P)		
performance (P)		
adaptability (P)		

Table 2 Potential quality factors

The two main concepts in the present approach to validation are requirements and rationale. These are very closely related to each other, since the exactness of the requirements limits the number of alternative solutions and at the same time the need for justification. They are described in more detail in Tervonen (1987). We shall consider next the role of rationale in the present validation approach.

PRE-REVIEWING AND JUSTIFICATION BY MEANS OF RATIONALE

According to the distinction between pre-reviewing and justification within validation (cf. Figure 7), rationale was also divided into "quality checklist" and "because of" types. The knowledge utilized in these forms of validation can be found from the literature or from company experiences and can be classified into quality metrics knowledge, method-independent knowledge and method-dependent knowledge. The latter two types can be utilized in both forms of rationale, whereas quality metrics is relevant only to "quality checklist" rationale.

The domain-specific knowledge type is not depicted in Figure 7 because it is included in the three other knowledge types at the physical level. At the logical level it would be reasonable to separate it out as an individual type of knowledge.

The relevant quality metrics are selected separately for each project by first fixing the relevant quality factors and then refining them into criteria, prioritizing them and selecting relevant criteria. Quality metrics in this context do not really measure or evaluate quality but consist of questions and recommendations which are important for checking the descriptions of the software design. These questions and recommendations are used in the literature as review checklists. For this reason the present pre-reviewing by means of a quality checklist partly shifts the responsibility for the reviews onto the designer. As an example of pre-reviewing and justification we consider SEM description (Software Environment Model) at the D level.

Only some of the questions and recommendations in the literature (e.g. Evans and Marciniak 1987) are directly applicable, in this case to the pre-reviewing of a SEM description. The Accuracy criteria (criteria for the Reliability factor) is one example of a readily applicable checklist, for in questions of that criteria only the object must be replaced (i.e. questions consider now task characteristics).

Accuracy:
1. Have all the accuracy requirements been checked at the level of individual tasks?
2. Is there a quantifiable requirement for the accuracy of task inputs, outputs and processing?

3. Have all the accuracy requirements been budgeted to individual tasks?
4. Is the error analysis performed and budgeted to the task?
5. Are execution outputs within the tolerances?

No all the checklists apply so well, e.g. the checklist of Simplicity criteria (in Evans and Marciniak 1987), refining the factors Reliability, Maintainability, Expandability, Flexibility and Reusability.

Simplicity:
1. Is the design organized in a top down manner? Module flow should be from top to bottom.
2. Are all modules independent?
3. Do module descriptions include input, output, processing and limitations?
4. Does each module have a single entry and exit point?
5. Is the data base properly compartmented?
6. There should not be duplicative functions.
7. A programming standard should be established, with the requirement for a structured code.

These checklists are not relevant to a SEM description, and are replaced here with method-dependent knowledge (Ward and Mellor 1985)

Simplicity:
1. Are there physical constraints to cause multi-tasking?
2. Are there physical constraints to cause distortion between logical model and task decomposition?

Another example of method-dependent knowledge concerning SEM descriptions (Ward and Mellor 1985) is

Modularity:
Does the task decomposition correspond to the functional decomposition of the logical model?

As an example of method-independent knowledge concerning "because of" rationale, we can take certain task decomposition rules (Gomaa 1984)

Dependency on I/O:
Depending on input or output, a transform is often constrained to run at a speed dictated by the speed of the I/O device with which it is interacting. In this case, the transform needs to be a separate task.

Time-critical functions:
A time-critical function needs to run as a high priority and therefore needs to be a separate task.

THE OSAAJA PROTOTYPE

The approach defined here has been employed to build a prototype known as OSAAJA (means "real expert"), after the research project of the same name. From the validation

point of view the prototype provided a basis for applying our review method "in practice" and at the same time for refining and clarify the meaning of the concept rationale. The prototype has been implemented with a Symbolics workstation and the frame based system KEE[1]. The present prototype is based on the model editor (Haataja and Seppänen 1986), which supports RT-SA/SD (Real Time - Structured Analysis/Structured Design) methodology, and also performs some consistency checking based on the RT-SA/SD methodology. The main idea of the OSAAJA prototype is to make the design decision situations (at all description levels) visible, to demand a rationale from the designer for each decision and to record these in connection with the design descriptions (in this case in connection with the RT-SA/SD descriptions). In testing the prototype we have used D level descriptions and especially SEM (Software Environment Model) descriptions. The main reason for this focus was our belief that the knowledge concerning this model should be obtainable in the easiest way.

The user interface of the prototype was implemented using KEE windows, the model editor or the Entity Relationship template (Figure 8). The main components of the pre-reviewing and justification process are represented in this template. If pre-reviewing or justification is commenced in the model editor, the name of the object (e.g. transformation or flow in a Data Flow Diagram) is presented in the instances box and the user can type in the name of the decision. The solid line in Figure 8 represents the essential components of the rationale. This rationale can include a textual description and/or causal factors, which can be a selection of documents (e.g. descriptions at the specification level), preceding decisions, external requirements and quality requirements. Activation of the documents box in the ER template allows one to fix the reference to the existing descriptions and elements of existing descriptions, and activation of any other causal factor box allows one to add/remove decisions or requirements fixed to the specific rationale.

---

[1] KEE = Knowledge Engineering Environment is a trademark of IntelliCorp

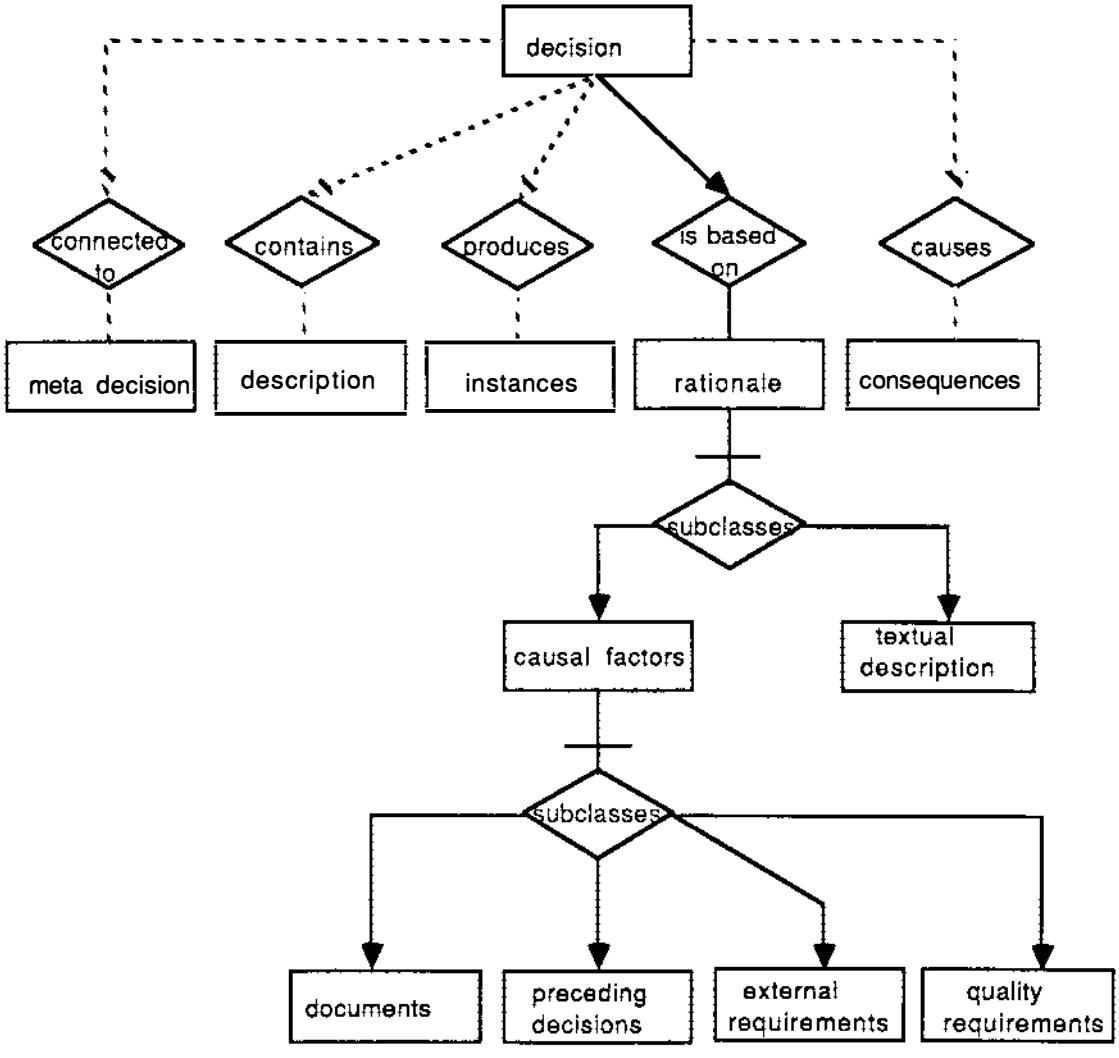


Figure 8 The ER template

The use of quality features and rationale in pre-reviewing and justification is a two-step process. First we make the necessary preparations at the beginning of the project (i.e. select the relevant quality factors and adjust the quality criteria) and then we start the design work. Once the relevant factors have been selected it is possible to adjust and prioritize the set of quality criteria. Figure 9 depicts a situation in which the user has selected three criteria (Modularity, Simplicity and Accuracy). These criteria and the checklists are utilized to pre-review the SEM description (an example of the questions and recommendations forming the checklist as presented before in this paper, is given in Figure 10). These checklists are connected to the description in Figure 11. Method-independent knowledge (Dependency on I/O) is utilized in individual decisions and connected to the decision in the same way (Figure 12).

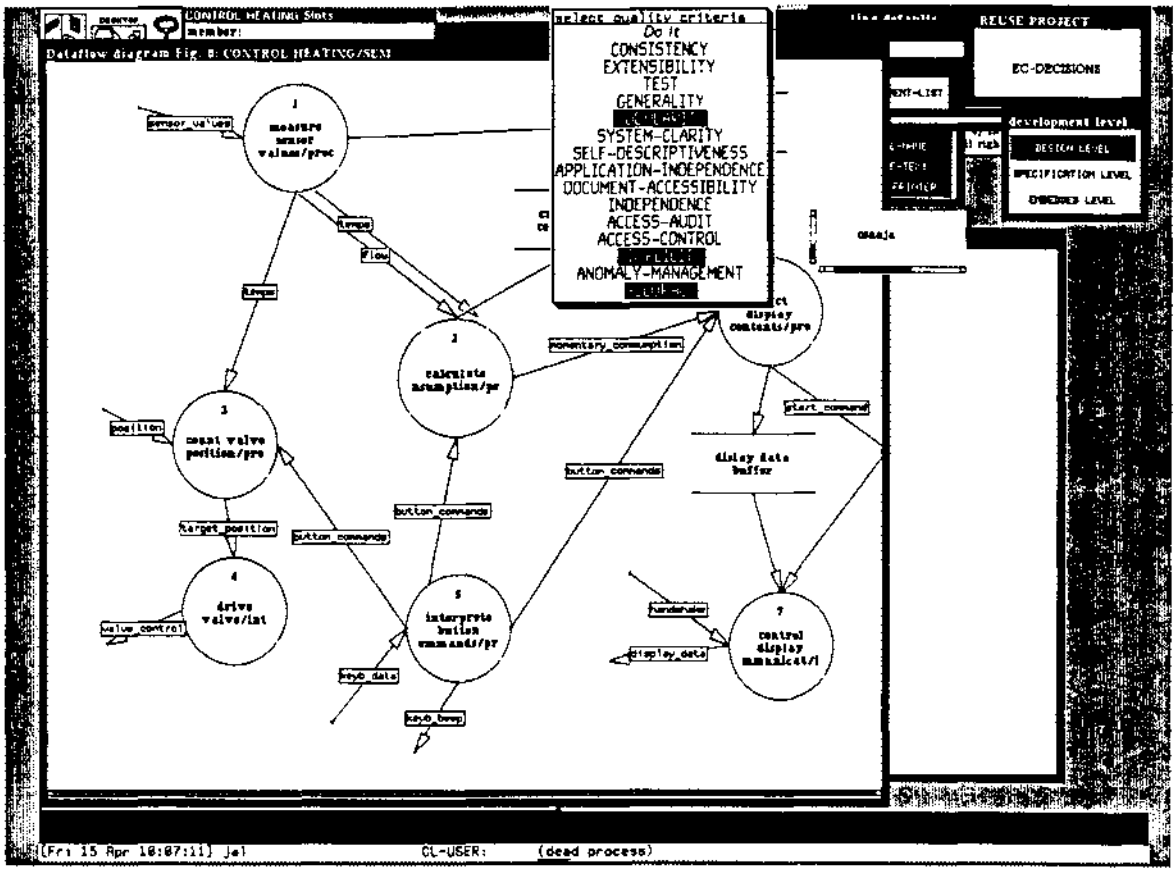


Figure 9

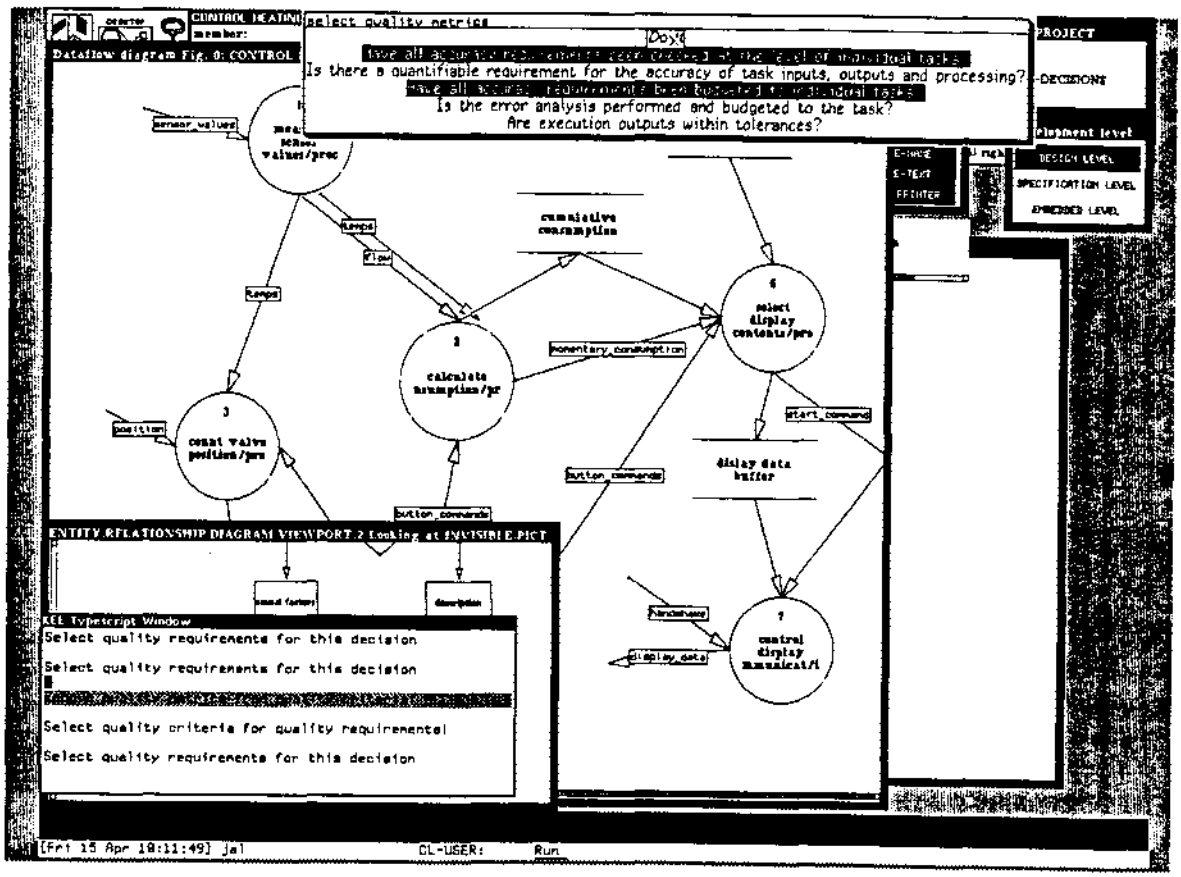


Figure 10

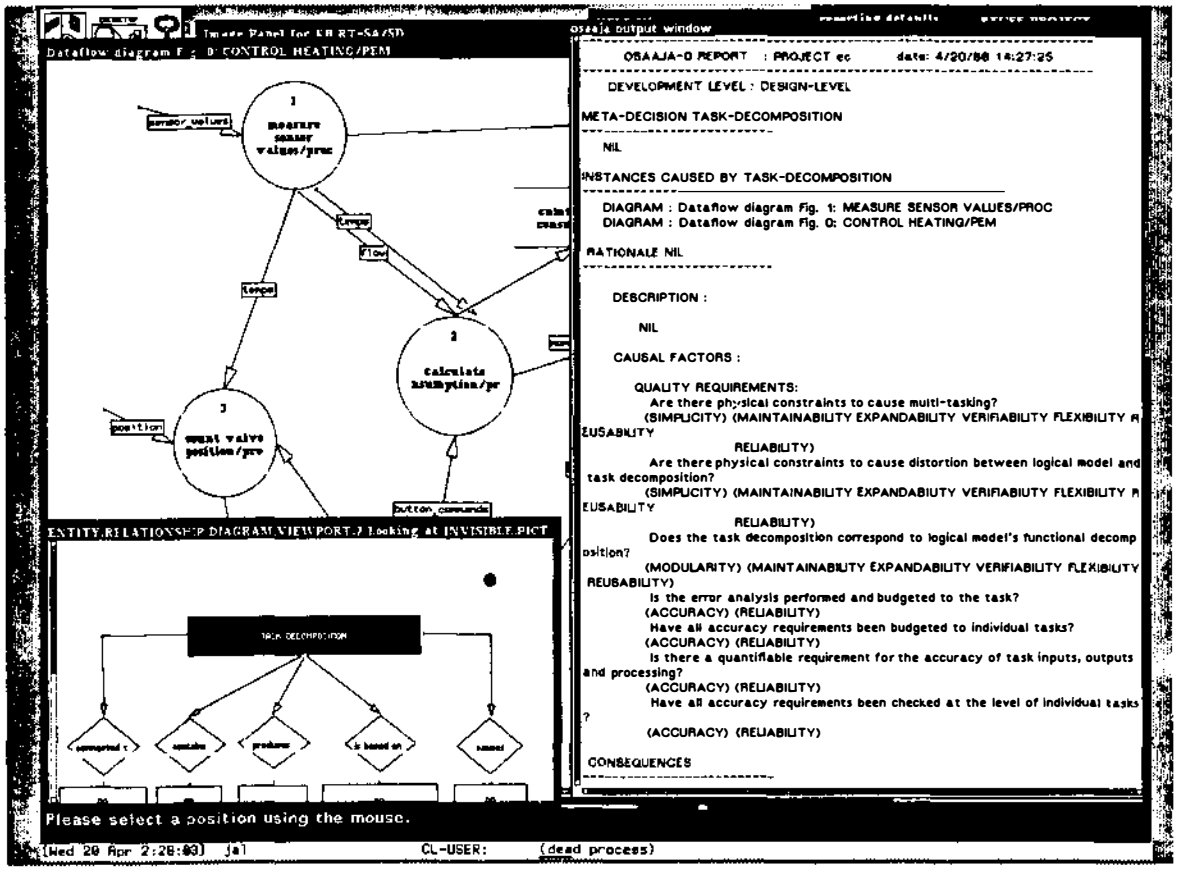


Figure 11

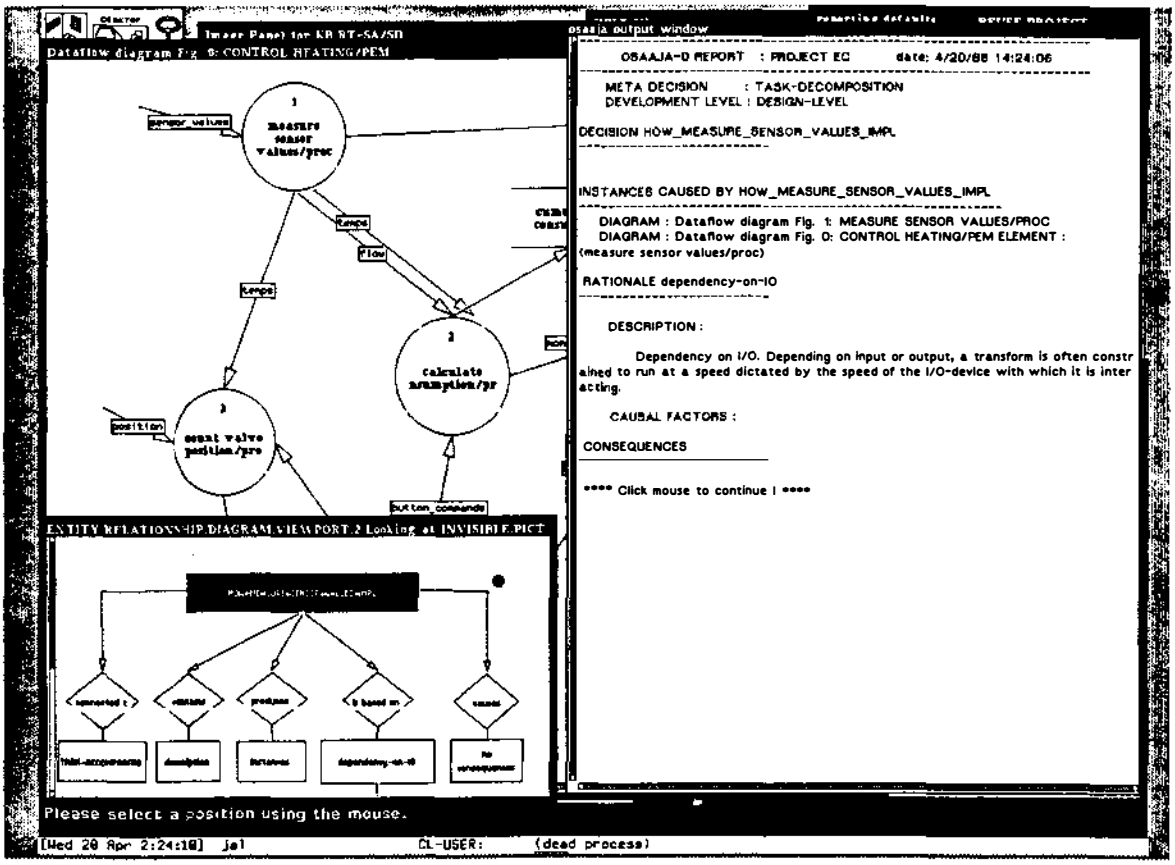


Figure 12

CONCLUSIONS

The conclusions to be derived from this paper can be divided into two parts as recards the theoretical framework developed here and the prototype developed in co-operation with the OSAAJA project.

The main purpose of the present fairly broad discussion of validation is to make all the characteristics of software validation visible, even though the focus is on informal reviewing. Especially we emphasize the primary role of validation in assessing the design descriptions at different levels which means that the software and its descriptions must be assessed against customer needs. Dynamic analysis (e.g. by means of reachability analysis), which is usually a fairly popular validation technique in the case of embedded software, is passed over in this study. Although correct behaviour of the embedded software is a core validation characteristic, we need, in this case too, a more common and wider theory to keep our hands on all the strings.

A variety of techniques are available for the validation of embedded software, and our validation by pre-reviewing and justification is just one instance of an informal inspection/walkthrough technique. As stated earlier in this paper, the difference between conventional reviewing and the present pre-reviewing is that the former is executed by a team and at a scheduled time, whereas the latter is executed by the individual designer in connection with the design process. Because the rationale is recorded in both pre-reviewing and justification, it is possible in the review situation to check (1) What have been the designer's reasons for the selection of a specific alternative?, (2) What checklists has the designer already looked through?, and (3) What has been the designer's rationale in individual design decisions? Although we have not discussed the reuse of designs included with decisions and rationale, we can with good reason argue that reuse is the aspect that makes the extra work profitable. We agree that the implementation of the prototype is unquestionably clumsy, but it is still a useful tool for acquiring the knowledge used as rationale in validation.

ACKNOWLEDGEMENTS

This paper is in many respects the result of co-operation in the CONSE and OSAAJA projects carried out as a part of the Finnish Programme for Research and Development in Information Technologies 1984 - 1988 and financed mainly by the Technology Development Centre of Finland. The final preparation of the paper was also financed by the Ministry of Education through its doctoral programme in data processing techniques.

The author wishes to acknowledge Juhani Iivari for inspiring conversations during the preparation of this paper and for his work as the insightful mentor of my

study, and also thanks the research teams engaged in the CONSE and OSAAJA projects.

REFERENCES

Boehm B.W., 1984, Verifying and Validating Software Requirements and Design Specifications, IEEE Software, vol 1, no 1

Brinch-Hansen P., 1973, Concurrent Programming Concepts, ACM Computing Surveys, vol 5., no 4

Evans M.W.,and Marciniak J.J., 1987, Software Quality Assurance and Management, John Wiley & Sons

Fagan M.E., 1976, Design and Code Inspections To Reduce Errors in Program Development, IBM Systems Journal, vol 15, no 3

Gilb T.,and Krzanik L., 1987, Design by Objectives, ACM SIGSOFT Software Engineering Notes, vol 12, no 2

Glass R.L., 1980, Real-Time: The Lost World of Software Debugging and Testing, Communications of the ACM, vol 23, no 5

Gomaa H., 1984, A Software Design Method for Real-Time Systems, Communications of the ACM, vol 27, no 9

Haataja R.,and Seppänen V., 1986, Support of Specification of Embedded Systems. A Practical Knowledge-based Approach, in Proceedings of Conference on Artificial Intelligence, Marseille, France

Hausen H.L., 1987, An Effectively Instrumentable Life Cycle Model, in Microprocessing and Microprogramming 21, North-Holland, Amsterdam

Iivari J., 1987, A Hierarchical Model for the Software Process: notes on Boehm's spiral model, ACM SIGSOFT Software Engineering Notes, vol 12, no 1

Iivari J., Koskela E., Ihme M.,and Tervonen I., 1986a, A Hierarchical Metamodel for Embedded Software: the embedding level as a mechanism for requirements analysis, in Barnes D., Brown P., (eds.), Software Engineering 86, Peter Peregrinus Ltd, London

Iivari J., Koskela E.,and Tervonen I., 1986b, A Macro Model for Embedded Software Development, Working Paper, Institute of Information Processing Science, University of Oulu

Kitchenham B., 1987, Towards a Constructive Quality Model, Software Engineering Journal, vol 22, no 4

Tervonen I., 1987, The Validation of Embedded Software: a layered approach and its instance, in Proceedings of Fifth EFISS Conference, Roskilde, Denmark

Tervonen I., and Alanko J., 1988, The Quality Checklist as a Rationale of the Decisions, in Proceedigns of the First European Seminar on Software Quality, Brussels, Belgium

Turski W.M., and Maibaum T.S.E., 1987, The Specification of Computer Programs, Addison-Wesley

Valmari A., 1987, Reachability Analysis -Based Validation of Embedded Systems, in Microprocessing and Microprogramming 21, North-Holland, Amsterdam

Wallace R.H., Stockenberg J.E,and Charette R.N., 1987, A Unified Methodology for Developing Systems, Intertext Publications, Inc., McGraw-Hill, New York

Ward P.T.,and Mellor S.J., 1985, Structured Development for Real-Time Systems, Volume I: Introduction & Tools, Yourdon Press, New York

Yourdon, 1984, Structured Analysis for Real-Time Systems, course material, Yourdon, Inc

Zave P., 1982, An Operational Approach to Requirements Specification for Embedded Systems, IEEE Transactions on Software Engineering, vol SE-8, no 3

MULTI-PERSON PROJECTS AND CASE

Byron Miller, Director Research
KnowledgeWare Inc.

ABSTRACT

Mr. Miller will discuss the use of CASE tools in an environment where users are working together to build applications software, focusing on the actual problems encountered when building systems. First reviewing systems development methods, then defining problems and finally discussing strategies and solutions to help facilitate cooperative work and move the entire group toward an amenable solution.

BIOGRAPHY

BYRON D. MILLER is the Director of Research for KnowledgeWare. In that capacity he is responsible for products that are two or more releases into the future. Mr. Miller has held various other positions in KnowledgeWare including Director of Product Planning and Manager of Methodology. Prior to joining KnowledgeWare, Mr. Miller had been active in its user group and served as its chairman.

1977-1985 Mr. Miller worked for E.I. duPont Textile Fibers Department where he held various positions in computing ranging from systems analyst to supervisor. He participated in the development of several computer aided manufacturing systems as analyst and lead analyst, consulted in methodologies and databases, introduced the use of relational database systems and machines, and developed "DATAPEDIA", the Textile Fibers' relational data dictionary system.

1974-1977 Mr. Miller worked for the Department of Defense where he was an Electronics Engineer. While there he worked in software support for an advanced development area and developed a large automated test system.

Mr. Miller has a B.S. in Electrical Engineering and a B.A. in Music from Rutgers University.

Contact at: KnowledgeWare Inc.
3340 Peachtree Road, N.E.
Suite 1100
Atlanta, GA 30026

(404) 231-8575

I. Nature of System Development

A. The Methods
B. The Problem Domain
C. The Analyst's Work

II. Working In Groups

A. More Than One
B. Dividing The Problem
C. Putting The Pieces Together

III. Automated Assistance

A. The Typical Solution
B. Toward A Better Solution Today
C. Desires For The Future

I. A. METHODS

1. WHAT CONSTITUTES GOOD METHODS[1]

a. they make important things explicit.
b. they expose natural constraints facilitating some class of computation.
c. they are complete - say all that needs to be said.
d. they are concise - say thing efficiently.
e. they are transparent - we understand.
f. they facilitate computation.
g. they suppress detail.
h. they are computable by an existing procedure.

2. TODAYS METHODS EXAMINED

a. DATA

1) Codd gave all methods a big boost with his work on Relational db.
2) The 3+NF went a long way to formalization.
3) Work continues in semantic db's.

b. PROCESS

1) DeMarco et al., gave us a place to start.
2) Little progress as compared with data.
3) Needs much work -
   - what is a process fragment
   - how are they associated
   - when is a process elementary
   - how do processes transform into procedures

c. UNIFIED THOUGHT[2]

1) Procedural methods
2) Deducive methods
3) Transformational methods
4) Inspection methods

---

1 P.H. Winston, *ARTIFICIAL INTELLIGENCE*, Addison-Wesley, Reading, Mass., p. 24.

2 C. Rich and R.C. Waters, "Automatic Programming: Myths and Prospects", *Computer*, Vol. 21 No. 8, August 1988, pp. 46-47.

I.B. THE PROBLEM DOMAIN

1. FRACTURED PROCEDURES

   a. Multiplicity of processors
      1) human
      2) computer
   b. Multiplicity of rules: fractured domains
      1) each implementation has only a subset of reality
      2) the subsets more than likely are conflicting (real world!)

2. EXCEPTIONS TO RULE

   a. Routine vs. exception is an artificial distinction
      1) Exceptions need handling outside the system.
      2) Exceptions are failures to include in the routine.
   b. "Retrospective illusion"
      1) a failure to remember the non ideal
      2) actual work is reconstructed as an instance of the ideal.

3. TOWARD UNIFICATION

   a. Need flexibility
      1) Fractured pieces of knowledge must be caputed as they are obtained.
      2) Conflicting information must be allowed.
   b. Need a rigorous structure
      1) Must be able to classify the pieces in order for unification to ever take place.
      2) Must have a conflict resoultion mechanism.
   c. Structure may be arbitrary as long as it is well understood.
      1) Arbitrary not equal ad hoc.
      2) Understood means a shared understanding.

I.C. THE ANALYST'S WORK

1. LIVING IN A FOREIGN LAND

   a. rarely does an analyst start out knowing much about the area under study.
   b. knowing too much may cause the analyst to suffer from retrospective illusion.
   c. knowledge is never complete.

2. PUZZLES WITH NO PUZZLE BOX

   a. the pieces don't come in one box with a picture; they are sought out.
   b. the pieces have rough edges.
   c. there may be conflictory pieces for the same slot.
   d. the pieces have cross puzzle dependencies.
   e. there are often several good solutions.
   f. pieces are often missing.
   g. the pieces change over time.
   h. the best solution is almost always a judgement call.

3. REFINEMENT AND EXTENSION - The ongoing work

   a. Refinement - knowing the problem in more detail.
   b. Extension - knowing the problem in a larger scope.

II.A. MORE THAN ONE

1. OFTEN THE ONLY WAY:

   a. when there is more than one analyst can analyze.
   b. when one analyst doesn't have enough background in all the facets of business that is involved.
   c. when one analyst doesn't have knowledge of all the techniques that are required.

2. MANY HANDS MAKE LIGHT WORK - SOMETIMES

   a. assumes each is doing own part.
   b. assumes own part can be defined.
   c. assumes parts are cooperative.

II. B. DIVIDING THE PROBLEM

1. BY STYLE

   a. anarchy - everyone does his/her own thing.
   b. muted anarchy - everyone does his/her own thing for a while then sharing occurs and the open minded are influenced.
   c. network - peer groups divide work, coordinate, resolve at intervals.
   d. modified hierarchy - a coordinator aids in dividing the work, coordinating and resolving issues; most work in done in peer groups.
   e. hierarchy - a storng boss divides work, coordinates individuals and resolves the issues (the boss knows what is right!)
   f. it all depends....

2. BY THE METHOD

   a. functional decomp
   b. event
   c. object
   d. output to input
   e. data
   f. you got one?

3. EVEN PROPERLY DIVIDED, THERE IS THE PROBLEM DOMAIN

   a. seeking your assignment, you get other answers.
   b. What will you do, what will you do?!
      1) ignore
      2) tread
      3) propose

II. C. PUTTING THE PIECES TOGETHER

1. EACH ANALYST UNDERSTANDS PART OF THE PROBLEM

   a. it is a given when there are many hands.
   b. it is a given because of fractured domains.

2. THE BEST SOLUTION IS THROUGH THE INPUT OF MANY

   a. the conflict must be resolved.
   b. hopefully the process will be synergistic and not synthetic.
   c. the final understanding must be held by all.

3. FACILITATING THE COOPERATION IS THE HARD PART

   a. there must be mutual awareness of the known.
   b. the unknown must be obvious.
   c. unaware tromping must not be allowed.

III. A. THE TYPICAL SOLUTION

1. A VARIATION ON THE TRANSACTION PROCESSING MODEL.

   a. has a multiuser db.
   b. last commit is current truth.
   c. may have archival.
   d. seldom has real versions.

2. PROBLEMS

   a. conflicting states are not retained.
      1) subtlety of truths are lost
      2) exceptions disappear
   b. there can be no conflict resolution

3. RATHER THAN HELPING,THE TYPICAL SOLUTION BURIES OR THROWS AWAY MUCH OF WHAT HAS BEEN GARNERED.

III. B. TOWARD A BETTER SOLUTION TODAY

1. IT BEGINS WITH STORING MULTIPLE AND SOMETIMES CONFLICTING SOLUTIONS TO THE SAME PROBLEM

    a. requires an object orientation.
    b. requires multiple descriptions of same object.
    c. requires retention of multiple states of refinement and extension.
    d. requires integrating and non-integrating views.
    e. heritage needs to be retained.

2. IT'S AIDED BY ANALYSIS TOOLS AND SIMULTANEOUS DISPLAYS

    a. tools must be able to identify similar pieces of problem domains by analyzing the object descriptions.
    b. simultaneous display of alternatives and conflicts is necessary.
    c. selection of objects, extended objects, and implied involved objects is necessary.

3. IT'S FURTHERED BY RULE BASED COMBINATIONS OF CHOSEN PIECES

    a. combinations must be consistent with parts.
    b. combinations must be consistent with rules of manual entry.
    c. combinations must make use of the extended semantics of the involved objects.

III. C. DESIRES FOR THE FUTURE

1. BETTER METHODS

2. BETTER TRAINING

3. BETTER TOOLS

4. BETTER SUPPORT

Documentation

"Data on the Use of Stepwise Redefinement Approach"
Pierre N. Robillard and Daniel Coupal, Ecole Polytechnique, University of Montreal, Canada

"Using Formal Specification as a Documentation Tool: A Case Study"
Edward G. Amoroso and Jonathan D. Weiss, AT&T Bell Laboratories

DATA ON THE AUTOMATION OF PROGRAM'S COMMENTS

Pierre N. Robillard and Daniel Coupal
Software Engineering Laboratory
Ecole Polytechnique de Montreal
Montreal, Quebec, Canada

ABSTRACT

This paper addresses the problem of source code documentation. Source code documentation has two components: the description of the statement and the description of the steps required to go from the specification in natural language to programming language statements. The model we are proposing provides a different mechanism for each form of documentation. *Narrative* comments are used to describe programming language statements. *Operational* comments are used to outline the steps taken to reach the programming language level of understanding.

The operational comments are instantiated by refinements. This is the modeling of the stepwise refinement approach. A tool called is used to implement and automate this approach. Data from three medium-sized projects are presented. Results show that this approach provides a systematic and coherent mechanism for producing source code documentation. Productivity is increased and maintenance efforts are greatly reduced.

AUTHORS

Pierre N. Robillard is a professor of software engineering and the director of the software engineering laboratory at Ecole Polytechnique de Montreal. His current research includes software quality, software metrics, computing systems, formal specification, testing techniques and CASE.

Robillard received a PhD in electrical engineering from the Laval University of Quebec. He is also a member of IEEE and ACM.

Daniel Coupal is a candidate for a MS in software engineering at Ecole Polytechnique de Montreal, where he is an NSERC Fellow. His research interests are software metrics and statistical analysis.

Coupal received is BS from Ecole Polytechnique de Montreal. He is an IEEE student member.

Authors can be contacted at:

Department of Electrical Engineering,
Computer Science Section
Ecole Polytechnique,
University of Montreal
P.O. Box 6079, Station A
Montreal, Quebec
H3C 3A7
TEL: 514-340-4815

DATA ON THE AUTOMATION OF PROGRAM'S COMMENTS

Pierre N. Robillard and Daniel Coupal
Software Engineering Laboratory
Ecole Polytechnique de Montreal
Montreal, Quebec, Canada

1. INTRODUCTION

The introduction presents the model. The following introduces the SCHEMACODE tool used to produce data over the last four years. The third section shows results about this data. Finally, the conclusion outlines the features of this approach.

It is generally admitted today that source programs should be well documented. We define documentation as any text written in natural language or any graphical representation used to describe the implementation of a task in computer language. This documentation is expressed by comments in the source code program. It is mainly done by the programmer in the process of coding his program.

From the data of a recent study (1) we can extrapolate that source code documentation can have an essential economic impact on software maintenance budgets. The study showed that more than 40% of the defects were associated with implementation of the design into code. The defects were the result of communication and comprehension problems experienced by the individual programmer or between programmers. The study showed that:

- 53% of the defects were found in features that were previously functional;

- 10% of the defects were introduced as a result of corrections being made. However, this percentage is vastly understated, because we counted only those defects not detected by regression testing performed at the time.

- 64% of the defects were caused by incorrect logic errors, as opposed to missing or extra logic errors,

- 79% of the defects were caused by problems related to production and understanding.

We address the problem of source code documentation because it is a very special form of documentation. It is done by people who have not been trained to write documentation. Programmers are called upon to write documentation, but unlike analysts who have been trained to do DFD or Structure Chart documentation, programmers receive no special training. It is no wonder why most programmers dislike all aspects of documentation: producing it, reading it or modifying it.

There are few models or tools to help programmers document their programs. Quality control of source code documentation is hardly ever talked about. Nevertheless, source code comments are the only form of documentation embedded into the source code and it is often the only documentation for those who maintain programs.

It should be made clear to programmers whether they are to document the process of deriving code from specifications or document the details of their source code program.

The object of this paper is to present a model for source code documentation. The model is based on a process of inference which enables programmers to derive code from function specifications (In this paper, the term *function* refer to a procedure or a subroutine). The model can be taught to programmers and is supported by a software editor. The quality of the documentation can be controlled and made to comply with standards related to software engineering literature.

During software development, we need to capture and trace the process of translating human specifications into machine readable instructions. Often the programmers themselves do not even know how they go from one level of refinement to another. But they should be aware of the steps involved. Figure 1 illustrates the documentation mechanism to be automated, that is the documentation produced by the stepwise refinement approach.

Comments in the source code are required for two reasons:

- to describe the meaning of a statement (WHAT)

- or to indicate how a group of statements is derived (HOW).

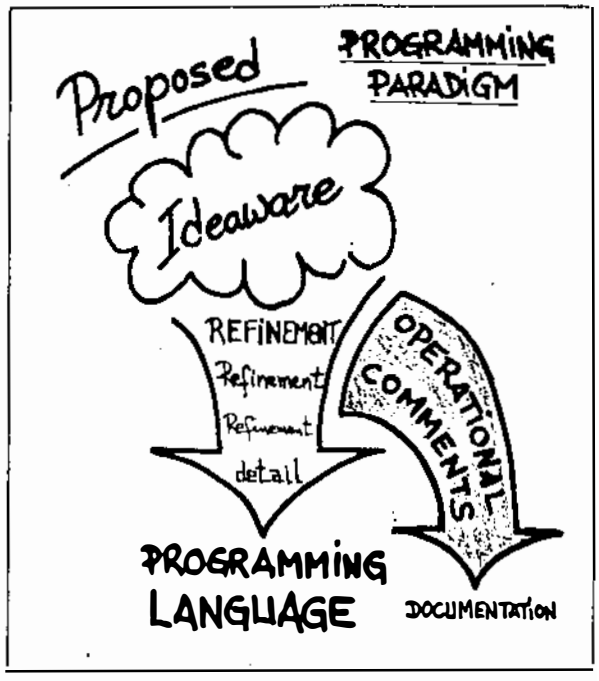


Figure 1

Proposed process for documentation

We are proposing a documentation method that identifies each type of comment and specifies when and how to use them. Comments used to describe the meaning of a statement (WHAT) are called *narrative* comments. Comments used to indicate the steps required before a group of statements can be derived (HOW) are called *operational* comments. The instantiation of an operational comment is a refinement. Operational comments represent the implementation of the stepwise refinement approach.

2. AUTOMATION OF THE DOCUMENTATION PROCESS.

Operational and narrative comments are part of a design methodology called *Schematic Pseudocode* (SPC). SPC also provides a schematic description of the control flow. SCHEMACODE is a tool that has been designed to automate the stepwise refinement approach and provide an automatic translation of the schematic pseudocode into the target programming language.

This paper is concerned only with the analysis of the documentation provided by the automated stepwise refinement approach. Readers interested in the details and the results of schematic pseudocode and its automation are referred to a previous article (2).

This section provides an example of program design using operational and narrative comments with the SCHEMACODE software tool.

SCHEMACODE is designed to generate and manage operational comments as well as SPC (Schematic PseudoCode). It has a high-performance, full-screen editor. It provides a natural mechanism for achieving higher levels of abstraction and thereby producing the necessary high-level documentation. This documentation is the most useful when retracing previous ideas in the maintenance phase. Moreover, the operational comments that are part of the SCHEMACODE abstraction mechanism are not normally intrinsic to programming languages or other software engineering techniques. Each operational comment describes an operation which will be specified later on and which is part of the stepwise refinement and top-down documentation processes.

SPC serves as a natural assistant to the programmer's thought process: it provides both a mechanism for the actual writing of programs (whereas most other techniques provide only a better way of describing programs), and, once the programs are written, high level program documentation.

Figure 2 shows the result of using the automated stepwise refinement approach with SCHEMACODE. The example is taken from a paper published by Bergland (3). It is a program to update an inventory from a transaction file.

The first step is represented by the refinement zero (R0) shown in the upper, left-hand corner of figure 2. Narrative comments, which are preceded by a dash, are used to identify the author and to describe the function to be implemented. The first two operational comments, which are preceded by a two-digit number, are used to define the included files and the variables.

Three other operational comments are used to specify the main steps: 03 OPEN FILE, 04 UPDATE INVENTORY and 05 CLOSE FILE. Refinement zero (R0), which contains five operational comments, is the main refinement.

Operational comment 03 is instantiated by refinement R3 (zoom in). This refinement is composed of two operational comments. Operational comment 06 OPEN TRANSACTION FILE is instantiated by the refinement R6. Refinement R6 is a refinement composed only of two programming language statements.

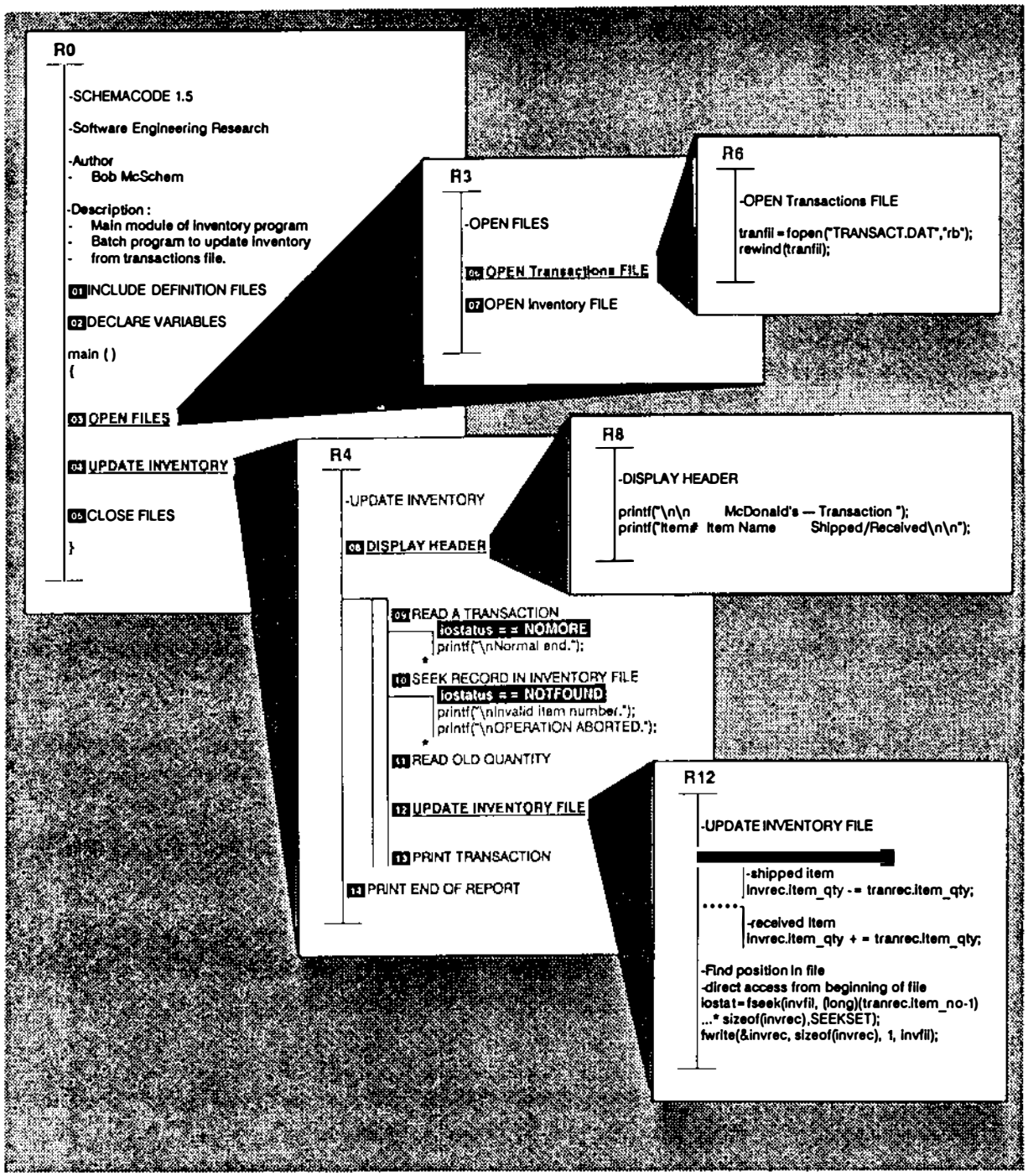


Figure 2

Refinement of ideas with operational comments

Operational comment 04 UPDATE INVENTORY (part of R0) is instantiated by the complex consolidation of refinements and executable statements. Refinement R4 includes a repetitive structure represented by schematic parallel lines, some executable statements and seven operational comments.

Operational comment 12 UPDATE INVENTORY FILE is instantiated by refinement R12 composed of one conditional structure and many programming language statements. Narrative comments (- shipped item, - received item, - Find position in file, - direct access from beginning of file) are used to document the executable statements.

A zooming feature enables the programmer to directly access any refinement. Operational comments are automatically converted into narrative comments once they have been instantiated by a refinement. A program is completed when there are no more operational comments to be instantiated.

This tool automatically supports a top-down approach based on stepwise refinements that are naturally implemented by means of the operational comments in SPC. SCHEMACODE automatically integrates all the documentation and guides the programmer along in such a way that the program is always properly structured. SCHEMACODE does not put any constraints on the features used in a programming language. It has been used mainly with C, FORTRAN, Pascal, dBASE III^{TM}, BASIC and COBOL.

3. DATA

The data presented in this paper are based on three projects developed in the C programming language. We refer to them as projects A, B and C. All the projects successfully resulted in reliable functional softwares. Projects A and B involved commercial software and project C involved software developed in a research laboratory. All analyst-programmers were professionals working full time on the projects. The use of the SCHEMACODE tool was mandatory. The analyst-programmers had solid working experience with this tool and only delivered software was analyzed.

The following table gives the size of the projects in terms of lines of code, total number of operational comments and number of functions. TOTAL represents the sum for the three projects.

PROJ.	LOC	OP	Func.
A	27520	2237	254
B	21670	2199	174
C	6277	488	69
TOTAL	55467	4924	497

Table 1

Statistics on projects

The projects were realized between May 1985 and December 1987.

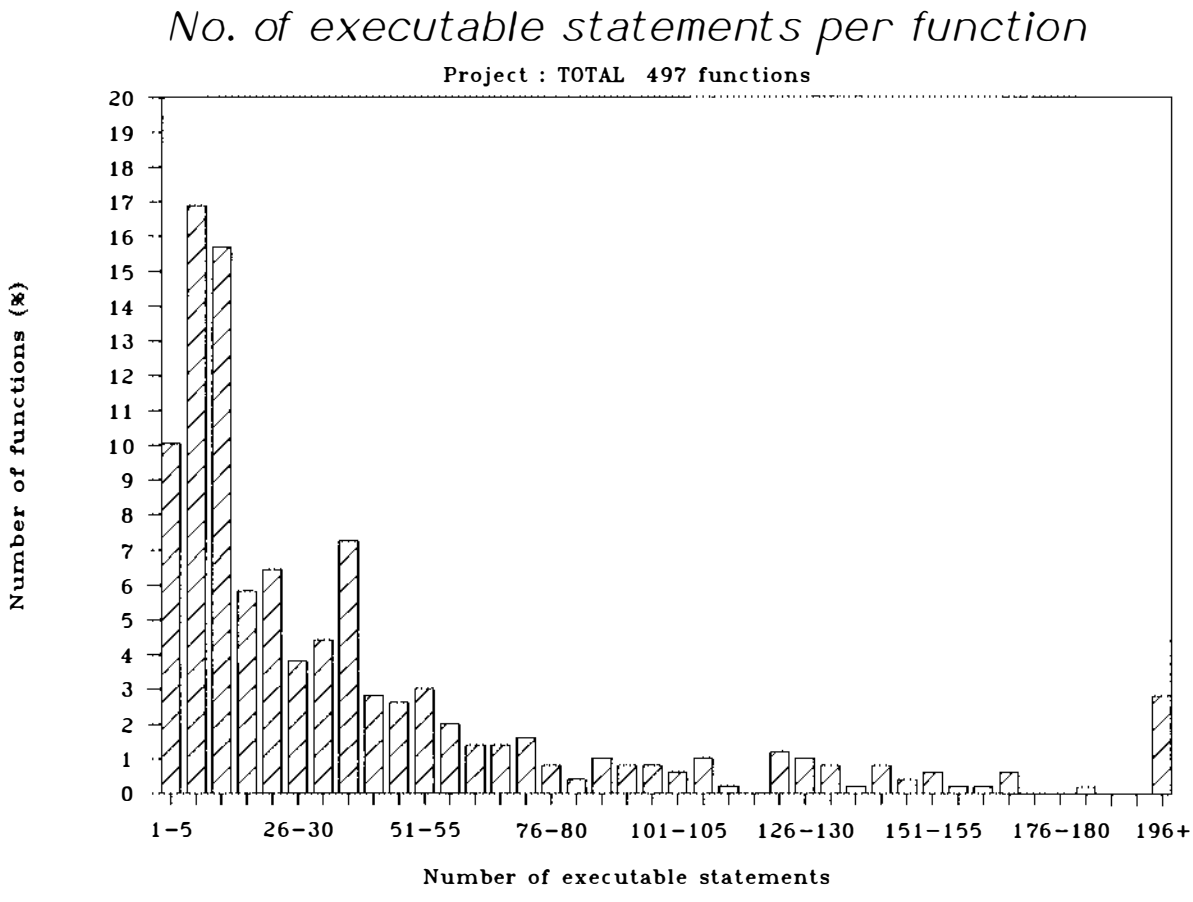


Figure 3

Number of executable statements per function

Figure 3 presents the number of executable statements per function. The rightmost column indicates that about 2% of the functions have more than 200 executable statements. The maximum found was 1476 followed by 585 executable statements.

We found that 75% of the functions have fewer than 40 statements, 50% have fewer than 20 statements and 6% have fewer than 5 statements. The latter are usually utilities. They are usually not part of the architectural design.

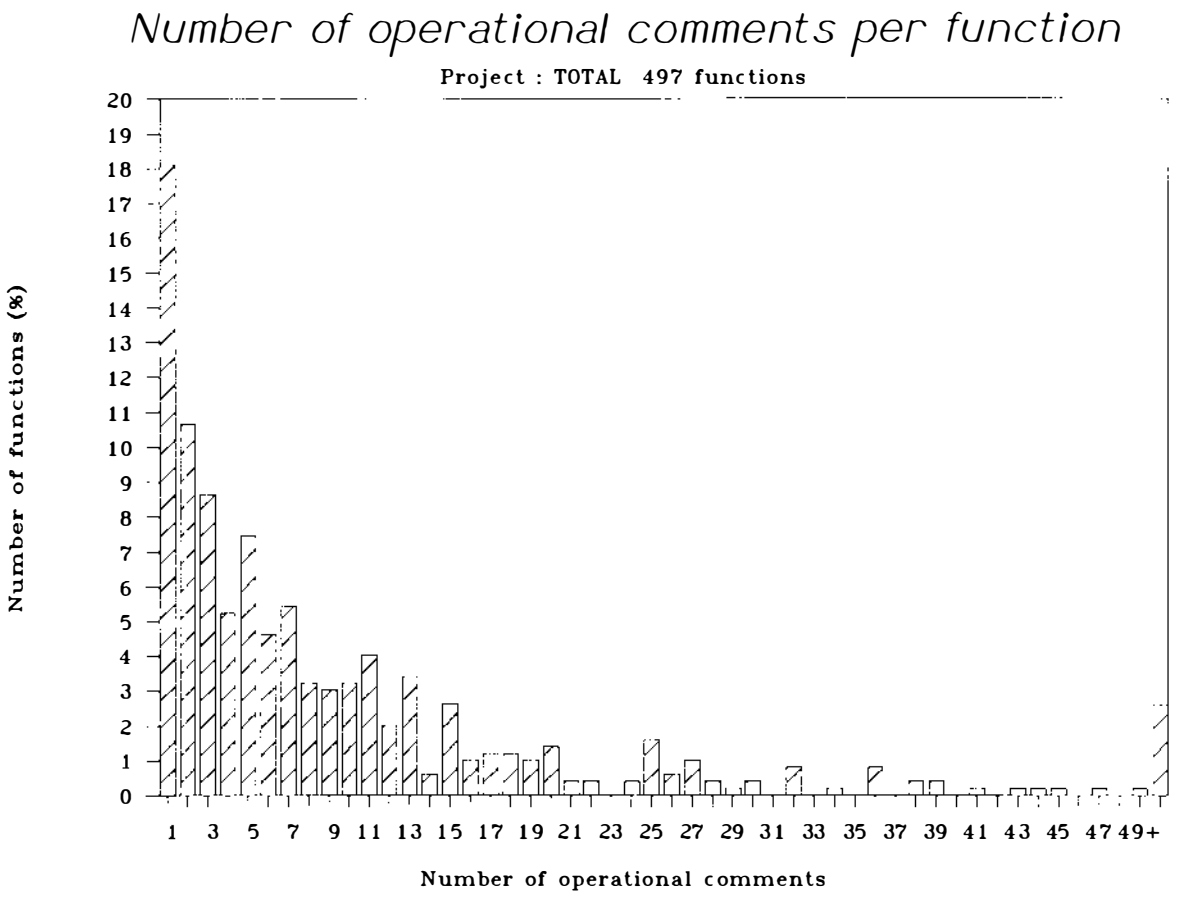


Figure 4

Number of operational comments per function

Figure 4 presents the distribution of the refinements or operational comment within a function. We observed that 18% (90) of all the functions have only a single operational comment. This occurs mainly when small utilities functions are developed as described before. The rightmost column includes the 12 functions that have 50 or more operational comments.

The maximum number of operational comments found in a single function is 79. These larger functions are characterized by a simple algorithm. For example, 3 functions are message handlers, 3 are screen managers, 6 are main functions. The programmer had used the refinement concept to group screens or messages in a hierarchical way. On the other hand, 50% of the functions have 5 or fewer operational comments, and 90% have fewer than 20 operational comments.

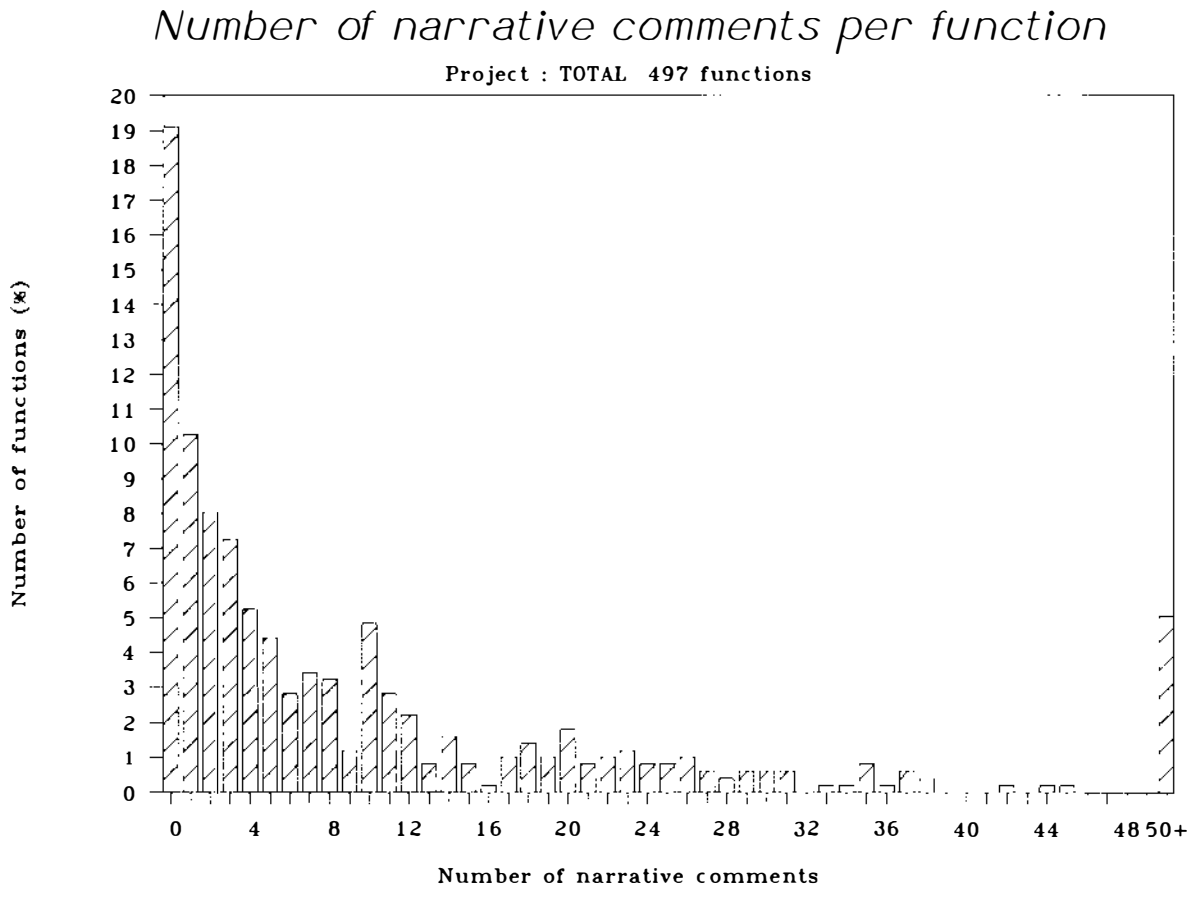


Figure 5

Number of narrative comments per function

Figure 5 shows the distribution of narrative comments per function. 19% of the functions do not have any narrative comments. This distribution looks like the distribution of operational comments (refinements). Thus, statements are referenced as well by narrative comments as by operational comments.

Because each operational comment refers to a task or a subset of a task in the code, the programmer has documentation that explains *what* is the portion of code by *narrative* comments and documentation that explain *how* this part of code is derived from by *operational* comments.

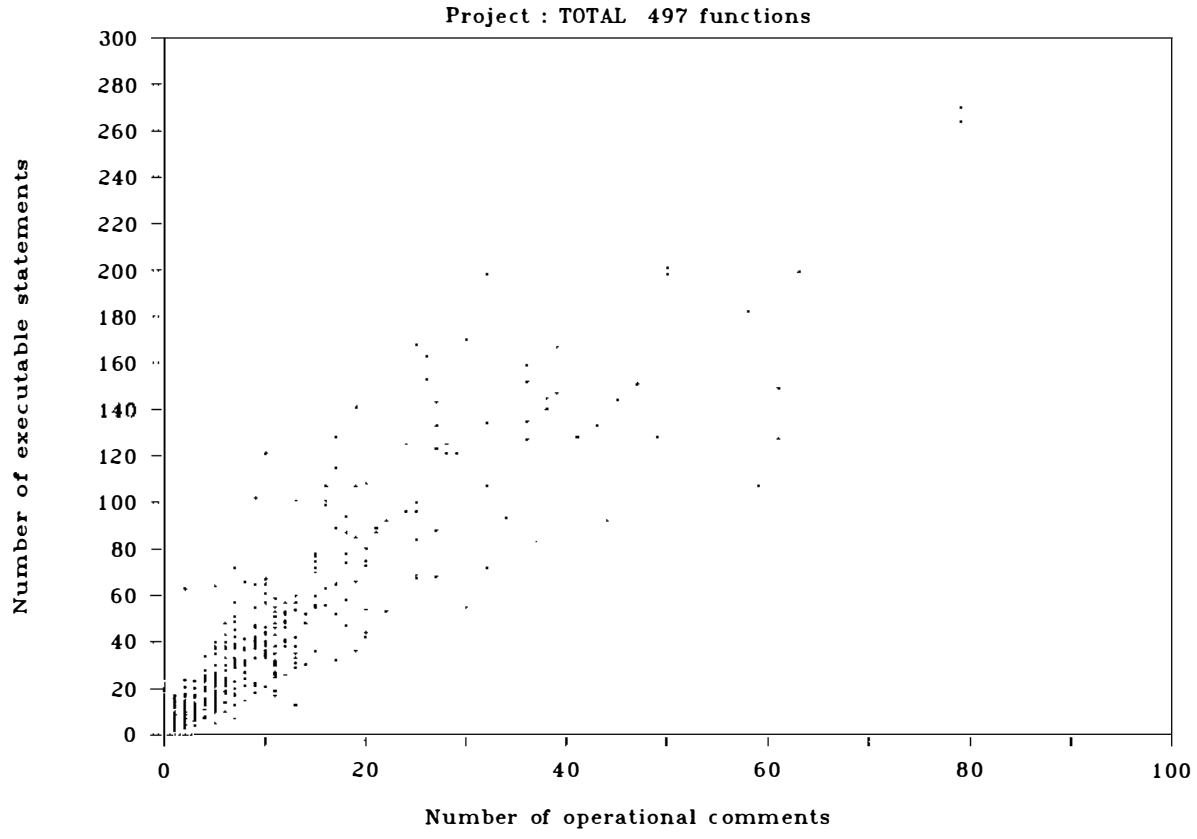


Figure 6

Operational comments and executable statements per function

Figure 6 shows the relationship between the number of operational comments and the number of statements in a function. The area in the lower, left-hand corner bounded by 60 executable statements and 20 operational comments contains 65% of the data.

In the seventies, some companies told their employees to write functions having less then 60 lines of codes. This decision was based on the absence of standard in software metrics. Considering the number of statements as a metric, we can see that these functions are usually explained by 10 to 20 refinements.

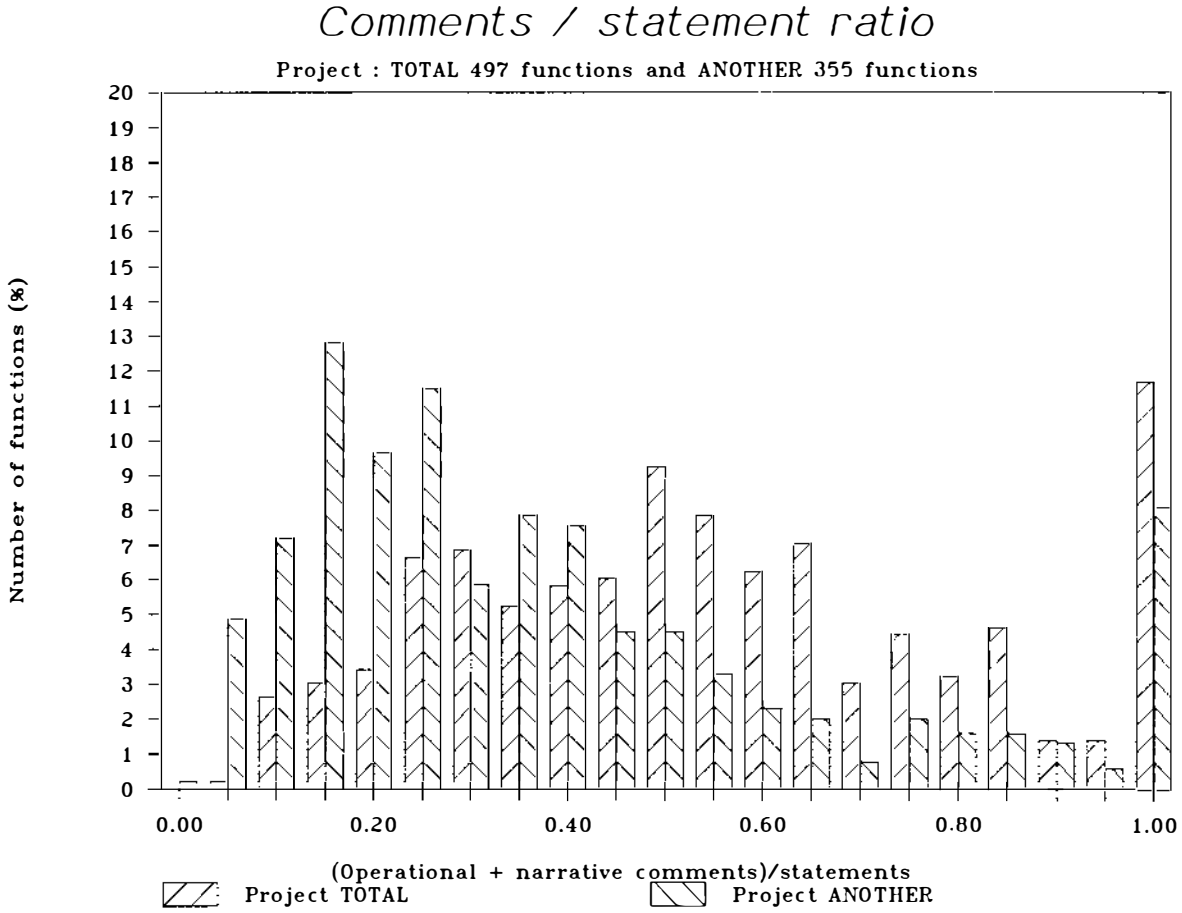


Figure 7

Comment ratio for 2 projects

Figure 7 shows that narrative comments are used in conjunction with operational comments to describe functions written with SCHEMACODE. We call total comments the sum of operational and narrative comments. In the project realized without SCHEMACODE, named ANOTHER, the total comments is the sum of all individual comments found in the source code.

Functions having ratios of 1.00 or more are represented by the rightmost column. These functions are the most documented.

Functions represented by the leftmost column do not contain any comment.

Since a programmer who uses SCHEMACODE must document each refinement step, we can conclude that the tool helps to build programs having a better comments distribution. The missing comments in the ANOTHER project are the ones describing how statements are derived (*operational* comments).

4. CONCLUDING REMARKS

The use of SCHEMACODE to automate and structure the documentation has an impact on both productivity during the development phase and the effort required during the maintenance phase. It is difficult to measure these figures accurately because there are just too many parameters over which there is no control. However, we observed the following:

- the SPC method is very easy to understand;

- those introduced to the method were all enthusiastic about it; besides the three projects described in this paper, the method was used in many other commercial and government environments;

- for project B, which was carried out for a large organization, productivity (measured in terms of the number of lines of code per person-month) was twice the standard figure for that organization; the other projects were for smaller organizations where no such data were available;

- development is exclusively interactive; programmers used operational comments with zoom in and zoom out features to build every piece of code; source code listings were never printed, used or even consulted (only the SPC representation on screen was used).

Projects A and B have been functional for two years and are now undergoing a maintenance phase.

- maintenance efforts are greatly reduced. Tests have shown that it takes as much as 50 times less time to identify a bug in a large program when the SPC representation is used.

This is due mainly to the fact that the documentation is built using a tree structure. The number of comments required to find any statement is always very small. Programs are not read sequentially but rather in the same way as they were designed. This traceable feature in the documentation is the major improvement provided by this approach.

Programmers have a tendency to develop small functions. The size of these functions is well within the constraints imposed by SCHEMACODE which represent a maximum of 100 refinements and 65,000 lines of code for the same function. The stepwise refinement approach prevents them from making very large multifunctional functions.

Function size may also be related to a previous study which revealed the magic number 7 ± 2 (4). With this approach, programmers are now aware of the number of steps they take in programming. They naturally try to keep this number small.

Littman, Pinto, Letovsky and Soloway (5) studied mental models in software maintenance and claimed that one of the programmers' most used strategy is to attempt, as soon as possible, to localize parts of program to which changes can be made that will implement the modification. Letovsky (6) found that programmers ask questions about:

- the purpose of an action or design;

- the way some goal of the program is accomplished.

The first question (what is doing this piece of code) is answered by *narrative* comments and the second question (how this idea is implemented) is answered by *operational* comments that describe the idea.

Following CASE tools, we present this model of documentation and programming techniques as the same way that humans think when they try to solve problems in a sequence of actions.

ACKNOWLEDGMENTS

We thank our colleagues André Beaucage, Alain Grenier and Jean-Bernard Trouvé from the Software Engineering Laboratory of Ecole Polytechnique of Montreal and Greg Boone of CASE Research Corporation who was the principal reviewer of this paper. Their perceptive comments in the long discussions we had with them have been very useful.

We also thank Bell Canada Inc. for data and source code used for analysis.

This research was sponsored by the Natural Sciences and Engineering Research Council of Canada under the Grant A-0141.

BIBLIOGRAPHY

1. Collofello, James, Jefferey, S. and Buck, J. *Software Quality Assurance for Maintenance,* IEEE Software, Sept. 1987, pp. 46-51.

2. Robillard, Pierre - N. *Schematic Pseudocode for program constructs and its computer automation by SCHEMACODE,* Comm. ACM Nov. 1986, Vol 29, no 11, pp. 1072-1089.

3. Bergland, G. D. *A guided tour of program design methodologies,* IEEE Computer, oct 1981, pp. 13-37.

4. Miller, George A. *The magical number seven, plus or minus two: some limits on our capacity for processing information,* The Psychological Review, March 1956, pp. 81-97.

5. Littman, D. C., Pinto J., Letovsky, S. and Soloway, E. *Mental Models and Software Maintenance,* The Journal of Systems and Software, Vol 7, 1987, pp. 341-355.

6. Letovsky, Stanley *Cognitive Process in Program Comprehension,* The Journal of Systems and Software, Vol 7, 1987, pp. 325-339.

Using Formal Specification as a Documentation Tool: A Case Study

Edward G. Amoroso
Jonathan D. Weiss

AT&T Bell Laboratories

ABSTRACT

This paper describes an ongoing effort to use formal specification as a documentation tool for *System V/MLS*, an AT&T UNIX® System V-based multi-level secure operating system. It is shown that by carefully embedding formal specifications into existing prose documentation, one can substantially improve the quality of the documentation. An ambiguity that existed in the original System V/MLS security policy documentation is shown to have been identified, removed, and corrected as a result of this formal approach.

BIOGRAPHICAL SKETCHES:

Ed Amoroso is a Member of Technical Staff at Bell Labs in N.J. He is presently involved with the System V/MLS secure operating system development. Ed received his B.S. in physics in 1983 from Dickinson College in Carlisle, Pa., his M.S. in computer science in 1985 from the Stevens Institute in Hoboken, N.J., and is presently completing his Ph.D. thesis in computer science at the Stevens Institute.

Jon Weiss is a Member of Technical Staff at Bell Labs in N.J. Jon is also presently involved with the development of the System V/MLS secure operating system. Jon received his B.A. in mathematics and computer science in 1982 from New York University, and his M.S.E. in computer science in 1983 from the University of Pennsylvania.

Using Formal Specification as a Documentation Tool: A Case Study

Edward G. Amoroso
Jonathan D. Weiss

AT&T Bell Laboratories

ABSTRACT

This paper describes an ongoing effort to use formal specification as a documentation tool for *System V/MLS*, an AT&T UNIX® System V-based multi-level secure operating system. It is shown that by carefully embedding formal specifications into existing prose documentation, one can substantially improve the quality of the documentation. An ambiguity that existed in the original System V/MLS security policy documentation is shown to have been identified, removed, and corrected as a result of this formal approach.

1. Introduction

Several recent works have reported the use of formal specification techniques on complex industrial computer-based systems [Cohen 87, Hayes 87, Perry 87, Hester 81]. While these efforts have been successful in demonstrating the feasibility and practicality of using formal specification in the design, development, and verification of such systems, very little has been said about the potential for using formal specification as a tool for producing high-quality documentation.

This paper describes an ongoing effort at AT&T Bell Laboratories to apply formal specification techniques to the documentation of *System V/MLS*, an AT&T UNIX System V-based multi-level secure operating system [Flink 88]. Our approach has been to carefully embed formal specifications, as appropriate, into existing System V/MLS prose documentation. The resulting combination of prose and mathematics has proven to be a highly effective means for communicating information about the system to users, developers, and system engineers. In addition, as will be shown, an ambiguity in the original System V/MLS prose documentation was identified, removed, and corrected as a result of the formal approach.

2. Background and Overview

Although the goal of this paper is to report our experience using formal specification as a quality-enhancing documentation tool for System V/MLS, our work had originally focused on formal specification as a baseline for the formal verification process. We were especially interested in the verification process since the DoD "Orange Book" [DoD 85] includes requirements for formal specification and verification at several security certification levels. We were familiar with the well-known criticisms of the verification process [DeMillo 79], and our own experience with at least one automated verification system [Good 84], tended to support the criticisms. We therefore decided that an objective study of the specification and verification process for practically-sized systems, would help us to deal with the assurance requirements in the "Orange Book" most effectively and productively in the future.

Since many of our colleagues were not familiar with the technology of formal specification and verification, it was important for us to demonstrate our work and our results in a tutorial manner. Using System V/MLS as a first example was an important

part of this strategy. Since our colleagues were already familiar with the details of the system, the amount of intellectual effort needed to understand the specifications was reduced considerably. As it turned out, our resolve to stress a highly tutorial approach turned out to be one of the most significant aspects of our work.

Our investigation began with the formal specification of the System V/MLS security policy. As in a recent government benchmark study [Kemm 86], several formal specification frameworks were used including Gypsy [Good 84], Ina Jo [Kemm 80], Z [Sufrin 82], and SAM [Amoroso 85]. As the formal specifications began to emerge, it became clear that although the notations and approachs were different, the following characteristics were uniformly present in each of the specifications:

- *Rigor*: All claims made in each specification could easily be traced back to explicit formal definitions.
- *Accuracy*: Extreme care was taken to remove any imprecisions, ambiguities, or errors from the specifications.
- *Completeness*: The boundary conditions of each specification were subjected to careful, logical reasoning.
- *Clarity*: Since we were concerned with producing tutorial explanations, the resulting specifications were succinct and to-the-point.

The presence of these characteristics in each of the specifications (which came as no great surprise), prompted us to go back and reexamine the existing prose documentation for the System V/MLS security policy. The quality of this existing prose had always been considered adequate and acceptable by the developers, engineers, managers, and customers of System V/MLS. The descriptions were thought to be understandable, manageable, and exactly what our customers expected (and, in some cases, demanded). However, upon careful reassessment, it became clear that the formal specifications were superior to the prose descriptions in terms of the above-mentioned characteristics (rigor, accuracy, completeness, and clarity).

There appeared, at first, to be two possible ways to incorporate the improvements into the prose descriptions: we could rewrite the prose, or we could throw away the prose and use the formal specifications as the documentation. Both choices had obvious disadvantages. Simply trying to rewrite the prose, for example, seemed futile since it had been carefully written over a long period of time, and represented as good an effort as we could reasonably expect. Furthermore, prose has certain inherent limitations that cannot be avoided.[1] Using just the formal specification as the documentation and scrapping the existing prose was deemed totally impractical. There were certain concepts that were simply best expressed using prose. Also, as we have already mentioned, some customers would not react too favorably to such a format.

As one might expect, a compromise approach was eventually developed that incorporated both the existing prose and the formal specifications. What was novel about the approach, however, was that the formalisms were not relegated to an

1 Case in point: The original Algol 60 report [Naur 60] is as good a prose document as one will find, and yet a large collection of problems surfaced [Knuth 67].

appendix, but were actually embedded right into the existing prose. This forced a careful reasoning about the content expressed in the prose, and, as will be shown, caused some of the prose to be completely reorganized and rewritten.

It should be recognized that although the formal specification framework used in this paper is based on the System Abstraction Methodology (SAM) [Amoroso 85], it has been clear to us throughout the project that the chosen specification framework is of less importance than an adherence to the methodology.

3. A Preliminary Example: System V/MLS Basic Concepts

In order to demonstrate our style of writing so-called "formalized documentation," an excerpt from the actual security policy description is provided below. The example is appropriate because it introduces several security policy concepts that are assumed in the main example in this paper. It is also appropriate because it is taken from the beginning of the actual policy description, and as such, includes a good deal of lead-in material. However, the excerpt is not a good example of our embedding approach because it had no prose counterpart in the original prose description. In the prose description, concepts were introduced as they were referred to, rather than in an explicit introduction. As will be shown again later, embedding the formal specification often forces a reorganization, a rewriting, or, as in this case, an addition to, the existing prose. It must therefore be kept in mind that this first example demonstrates our *style* of writing documentation, but leaves the description of our embedding approach to the next example.

It is also important to note at this point, that the result of our "formalized documentation" approach is really nothing new. Mathematicians have always expressed themselves using a combination of prose and mathematics. Unfortunately, however, most computer-related documentation is simply not written this way, and the notion of using a formal specification to improve an existing prose description is quite new.[2]

The following excerpt (shown in a slightly smaller point size), is taken directly from the formalized description of the System V/MLS security policy (the appendix provides a brief description of the specification notation):

> *System V/MLS Security Policy: Basic Concepts:*
>
> System V/MLS provides a secure environment in which all actions that take place are mediated. The security policy chosen to control these actions is based on several previous works, including the "Orange Book" [DoD 85], and the pioneering work of Bell and LaPadula [Bell 74].
>
> The two most basic entities in our framework are the *subjects* and the *objects* of the system. Actions occurring on the system must be initiated by someone or something. We refer to the initiator of such actions as a *subject*. It turns out that we can actually equate subjects to System V/MLS *processes* in our security policy. It must be recognized that *primitives* may be introduced in our description as appropriate. Primitives are to be interpreted as entities whose details are not relevant to the behavior being studied The set *processes* will be used as a primitive entity in the definition of System V/MLS subjects:

2. We refer the reader to our concluding remarks for references to similar works.

$$\text{subjects} = \text{processes}$$

Similarly, System V/MLS objects will be defined as the union of two primitive sets as follows:

$$\text{objects} = \text{files} \cup \text{directories}^{3}$$

Using subjects and objects, we may create a model known as a *reference monitor* which will determine whether particular operations attempted by subjects using objects are allowed. For instance, the reference monitor will determine whether a particular process can *read* from or *write* to some file. [4] Obviously, in order to implement such a protection policy, one must have some way of specifying the *sensitivity* associated with each subject and object. On System V/MLS this is done via a concept known as the *privilege*.

Each System V/MLS privilege has a *discretionary* and a *mandatory* component. The discretionary component is referred to as a *group* and the mandatory component is referred to as a *label*. Further decomposition reveals that a label consists of a hierarchical component known as a *level* and a non-hierarchical component known as a *category set*. Before we continue, let's describe this more formally. We will use the sets *levels*, *categories*, and *groups* as primitives in the description as follows: (Note that we denote the cross product of two sets A and B by the notation "A × B" and the power set of a set A by £ (A)).

$$\text{labels} = \text{levels} \times \pounds(\text{categories})$$
$$\text{privileges} = \text{labels} \times \text{groups}$$

At any given instant, a current operating privilege is associated with every subject and object on the system.[5]

$$\text{clear} : \text{subjects} \rightarrow \text{labels}$$
$$\text{class} : \text{objects} \rightarrow \text{labels}$$

In order to enforce a mandatory policy, it is necessary to define binary relations on labels. The two basic relations that are used are called *EQUALS* and *DOMINATES*. Recall that binary relations are simply sets of ordered pairs. Thus the EQUALS relation will simply be the set of ordered pairs such that the first component of the pair and the second component of the pair are exactly the same label. This is defined formally as follows (Note that a label L has a "levels" component and a "categories" component. We will use the shorthand notations "levels (L)" and "categories (L)" to refer to each of these components respectively).

$$((L_1, L_2) \in \text{EQUALS}) \text{ iff}$$
$$((L_1, L_2 \in \text{labels}) \wedge$$
$$(\text{levels}(L_1) = \text{levels}(L_2)) \wedge$$
$$(\text{categories}(L_1) = \text{categories}(L_2)))$$

---

3. We have only included in the definition of objects those primitives that are referred to in the paper. The full definition of System V/MLS objects includes interprocess communication mechanisms and processes.
4. In the full policy description, the reference monitor takes the directory in which a file resides into account as well.
5. For the purposes of this example, only the operating label is considered.

Notice that using this definition, one could easily prove that EQUALS is an equivalence relation (as one ought to expect)!

A second binary relation defined on labels is called *DOMINATES*. This relation relies upon the existence of an established *level* hierarchy. Recall that the first component of a label is a hierarchical level. The first requirement for a label L_1 to DOMINATE another label L_2 will be for the level component of L_1 to be hierarchically "equivalent to" or "greater than" the level component of L_2

When a label L_1 is hierarchically greater than a label L_2, we denote this fact by writing $L_1 > L_2$

Similarly, the symbols '$<$' and '$\geq$' denote "less than" and "greater than or equal to" respectively.

Recall further that the second component of a label is a categorical component consisting of a (possibly empty) set of categories. The second requirement for a label L_1 to DOMINATE another label L_2 will be for the category set of L_1 to be a "superset" of the category set of L_2

Thus we may define the DOMINATES relation formally as follows:

$$\begin{array}{c}((L_1, L_2) \in \text{DOMINATES}) \text{ iff} \\ ((L_1, L_2 \in \text{labels}) \wedge \\ (\text{levels}(L_1) \geq \text{levels}(L_2)) \wedge \\ (\text{categories}(L_1) \supseteq \text{categories}(L_2)))\end{array}$$

The above excerpt demonstrates the basic style used to write our formalized System V/MLS security policy documentation. Our next example will demonstrate our approach to improving existing prose descriptions via the formalism.

4. Original Reclassification Policy Prose Description

An excerpt from the original prose security policy description dealing with the rules for reclassification of objects is presented below. Presumably, if a System V/MLS user wanted to reclassify an object, then the description below would provide sufficient information on whether this was allowed, and if so, what conditions were necessary.

It will be shown in subsequent sections that upon formalization of this policy excerpt, an ambiguity was identified, removed, and corrected (the reader is invited to try to identify the ambiguity while reading the prose). In addition, the formalization process prompted a reorganization and rewriting of the prose description (beyond simply eliminating the ambiguity).

The following excerpt is taken directly from the original prose description of the System V/MLS security policy:[6]

System V/MLS Reclassification Policy:

The reclassification command, *chlev*, is a Trusted Computing Base (TCB) command that allows a user to change the label on a file or directory.

6. Several UNIX and System V/MLS concepts such as user and group IDs, mandatory read permissions, and object ownership are referred to without explanation. [Bach 86] provides a clear introduction to UNIX and [Flink 88] provides a similar introduction to System V/MLS.

Chlev uses the following access rules.

Case 1 (Unrestricted Reclassification): If

- the invoking process has effective user ID of "root," and
- the new object privilege (new label + current discretionary group) is defined on the system,

then the file or directory label is changed

Case 2 (Declassification): If:

- the invoking process has real group ID of "secadm," OR user ID equal to the object owner, and effective group ID of "secadm,"
- the invoking process has mandatory read permission for the file or directory,
- the specified new label is dominated by the current label on the file or directory, and
- the new object privilege (new label + current discretionary group) is defined on the system,

then the file or directory label is changed

Case 3 (Increased Classification): If

- the invoking process has user ID equal to the object owner,
- the specified new label dominates the current label on the specified file or directory,
- the new object privilege (new label + current discretionary group) is defined on the system,

then the file or directory label is changed.

It must be recognized that the above excerpt was part of a security policy description that was viewed as adequate and acceptable by the developers, engineers, managers, and customers of System V/MLS. There was no reason to believe that the description could be improved in any significant way.

5. Embedding the Formal Specification into the Existing Prose

Recall that our basic approach has been to embed formal specifications into existing prose descriptions with the intention of improving its quality. The excerpt below demonstrates a direct application of the approach for the reclassification policy description. Formal specifications were embedded directly into the prose description, with as little change to the prose as was possible.

It must be noted that some concepts had to be introduced explicitly in order to be used in the formal specification. For instance, the effective and real user and group IDs (EUID, UID, EGID, GID), had to be explicitly introduced so that the specification could make use of them. It is not, however, important that we provide all of these explicit definitions in our excerpt. Our purpose is to demonstrate the approach, and to this end, we will avoid any unnecessary detail.

System V/MLS Reclassification Policy:

The reclassification command, *chlev*, is a TCB command that allows a user to change the label on a file or directory.

Chlev uses the following access rules:

Case 1 (Unrestricted Reclassification): If

- the invoking process has effective user ID of "root," and

- the new object privilege (new label + current discretionary group) is defined on the system

then the file or directory label is changed

This rule may be described by the following event description:[7]

reclassify (p, o):
p ∈ process
o ∈ object
EUID (p) = root

¬ (class(o) = class(o´))

Case 2 (Declassification): If:

- the invoking process has real group ID of "secadm," OR user ID equal to the object owner, and effective group ID of "secadm,"
- the invoking process has mandatory read permission for the file or directory,
- the specified new label is dominated by the current label on the file or directory, and
- the new object privilege (new label + current discretionary group) is defined on the system,

then the file or directory label is changed

This rule may be described by the following event descriptions:

declassify_secadm (p, o):
p ∈ process
o ∈ object
GID (p) = secadm
clear(p) DOMINATES class(o)
---
class(o´) DOMINATES class(o)

declassify_owner (p, o):
p ∈ process
o ∈ object
UID (p) = owner
EGID (p) = secadm
clear(p) DOMINATES class(o)
---
class(o´) DOMINATES class(o)

Case 3 (Increased Classification): If:

- the invoking process has user ID equal to the object owner,
- the specified new label dominates the current label on the specified file or directory,
- the new object privilege (new label + current discretionary group) is defined on the system,

then the file or directory label is changed

7. Note that postcondition entities denoted with a tick (e.g. o´), refer to the value associated with that entity in the preconditions. The reader is again referred to the appendix for an informal description of the specification notation.

```
classify (p, o)
  p ∈ process
  o ∈ object
  UID (p) = owner
  ---------------
  class(o) DOMINATES class(o´)
```

It turns out that embedding the formalism into the prose allows one to reason more effectively about the behavior being described. Recall that one of the original purposes of the formal specification was to facilitate logical reasoning. Such reasoning applied to the above reclassification policy uncovered two important facts:

1. There was an ambiguity in the original prose description of the reclassification scheme.
2. The formal description of the reclassification rules could be reorganized into a more logical format.

These two facts prompted a complete reorganization and rewriting of both the existing prose description and the formal specification.

6. An Ambiguity in the Original Prose Description

The identified ambiguity is embedded in the description of the rules for unrestricted reclassification. Upon inspection of the above documentation, one may be led to believe that root permission is a necessary prerequisite to unrestricted reclassification. Thus, if a System V/MLS user wishes to reclassify an object upwards, downwards, or to an unrelated label, the seemingly obvious conclusion is that root permission is needed.

Consider, however, the following situation: a user who can operate with any security label defined on the system, owns a file and wishes to reclassify it in an unrestricted manner. If the user has real or effective group ID equal to "secadm" then unrestricted reclassification of that file is possible.

This situation, which might (or might not), be inferred from the prose documentation, is not consistent with the conclusion that root permission is required for unrestricted reclassification. Whether or not this represents a serious problem is not quite the issue. The issue instead is that high-quality documentation rarely suffers from such ambiguities. The best documentation has all of the aforementioned characteristics: rigor, completeness, accuracy, and clarity. Ambiguities are not consistent at all with these characteristics.

7. Reasoning About the Specified Behavior

Upon further inspection of the formal reclassification rules, there appeared to be a somewhat disorganized collection of rules (usually referred to as *events*). It must be understood that these events define so-called *state transitions*, and as such, define a collection of *reachable states* from a defined initial state. One can thus describe a particular system behavior by a *state-event sequence* in which the first element of the sequence is the initial state, the second element is an event *applied* to the initial state, the third element is the state resulting from this event application, and so on ([Parnas 77] calls such a sequence a *trace*).

With this in mind, one would like to define a collection of events that map logically to the behavior that is being modeled. In the case of the System V/MLS reclassification policy, this did not appear to be the case. Because we had used the prose description for the basic organization of our description, we really could not expect the logical flow to be any better than that of the prose. Certainly we had made the concepts clearer, and we had explicitly defined the notions being used in the description, but the basic organization remained the same.

For example, the necessary preconditions for "upward classification" were a subset of the preconditions for "downward classification" by owners. In general, two rules having similar or the same preconditions is not a problem as long as the events map logically to the behavior being modeled. In this case, the fact that these two seemingly unrelated operations had such similar preconditions called to our attention how confusing the whole organization was. It was determined that a complete reorganization would have to be done, and that implied a rewrite of the existing prose.

8. A Reorganized Reclassification Policy Description

The excerpt below describes the result of our reorganization and rewriting of the reclassification policy rules. The identified ambiguity has been removed and corrected, and the logical mapping of the events to the actual behavior being described is much improved.

It should be noted that in our reorganized description, the precondition that privileges be defined on the system before they are used has been pulled from the rules and made global to the section. We do not include this in our excerpt below.

System V/MLS Formalized Reclassification Policy:

The reclassification command, *chlev*, is a TCB command that allows a user to change the label on a file or directory. There are three types of reclassification operations to be considered: *declassification* from a label L1 to a label dominated by L1, *classification* from a label L1 to a label that dominates L1, or *reclassification* to an "unrelated" label. The purpose of this section is to provide the rules for such reclassifications.

Case 1: Declassification ("Downward"):

Declassification is defined as the explicit changing of an object label to a new label that is defined and is DOMINATED by the original label. Declassification of an object may occur whenever one of the following is true:

1. The invoking process has effective user ID of "root."
2. The invoking process has real group ID of "secadm" and mandatory read permission for the file or directory.
3. The invoking process has real user ID of "owner," effective group ID of "secadm," and mandatory read permission for the file or directory.

The following rules provide a more formal description of the above statements.

declass (p, o):
p ∈ process
o ∈ object
EUID (p) = root

---

class(o´) DOMINATES class(o)

declass_secadm (p, o).
p ∈ process
o ∈ object
GID (p) = secadm
clear(p) DOMINATES class(o)
---
class(o´) DOMINATES class(o)

declass_owner (p, o).
p ∈ process
o ∈ object
UID (p) = owner
EGID (p) = secadm
clear(p) DOMINATES class(o)
---
class(o´) DOMINATES class(o)

Case 2: Classification ("Upward"):

Classification is defined as the explicit changing of an object label to a new label that is defined and DOMINATES the original label. Classification of an object may occur whenever one of the following is true:

1. The invoking process has effective user ID of "root "
2. The invoking process has real user ID of "owner."

The following rules provide a more formal description of the above statements.

classify (p, o).
p ∈ process
o ∈ object
EUID (p) = root
---
class(o) DOMINATES class(o´)

classify_own (p, o).
p ∈ process
o ∈ object
UID (p) = owner
---
class(o) DOMINATES class(o´)

Case 3: Reclassification ("Unrestricted").

Unrestricted reclassification is defined as the explicit changing of an object label to a new label that is defined. This new label may DOMINATE, be DOMINATED by, or have no relation to the original label. Reclassification of an object may occur whenever one of the following is true:

1. The invoking process has effective user ID of "root."
2. The invoking process has real group ID of "secadm" and mandatory read permission for the file or directory.
3. The invoking process has real user ID of "owner," effective group ID of "secadm," and mandatory read permission for the file or directory

The following rules provide a more formal description of the above statements (note the empty sets of postconditions).

reclassify (p, o):
p ∈ process
o ∈ object
EUID (p) = root

¬ (class(o) = class(o´))

reclassify_2 (p, o):
p ∈ process
o ∈ object
UID (p) = owner
GID (p) = secadm
clear(p) DOMINATES class(o)

¬ (class(o) = class(o´))

reclassify_3 (p, o):
p ∈ process
o ∈ object
UID (p) = owner
EGID (p) = secadm
clear(p) DOMINATES class(o)

¬ (class(o) = class(o´))

9. Concluding Remarks

In summary, it has been demonstrated, via the System V/MLS security policy example, that formal specification techniques can be used as a tool for producing high-quality documentation. The benefits of more accurate and complete documentation are, for the most part, quite obvious. Too often, the documentation provided with a particular system is inadequate, and produces confusion for users and developers. By embedding a more rigorous model into the prose documentation, one provides the reader with a means for resolving any confusion.

As a concrete example of the potential benefit of a rigorous approach to documentation, consider the DoD National Computer Security Center (NCSC) evaluation community. It is their job to evaluate the secure aspects of a system by a variety of methods, including reading and evaluating the documentation. Embedding formal specifications into the documentation would make their evaluation task much easier, because it would remove much of the iterative and time-consuming question and answer sessions initiated by unclear and ambiguous documentation. In addition, if the formal specification is also being used as a baseline for verification, then the task of evaluating the verification is also made much easier.

It turns out that formal specifications with prose annotation are hardly an original notion. The programming research group at Oxford, for instance, has been producing specifications using a notation known as Z [Sufrin 82] that are heavily laced with careful prose explanations. However, their emphasis has been primarily

on the logical feasibility of producing such specifications, rather than on producing high-quality, usable documentation. As an example, consider the specification of a portion of the UNIX filing system presented by Sufrin and Morgan [Sufrin 84]. The emphasis in this work is on demonstration of the feasibility of rigorous, accurate UNIX specification. Clearly, their work is not intended as tutorial documentation.

An additional related work is Knuth's *literate programming* [Knuth 84], an approach to programming and documentation that encourages the programmer to embed a carefully constructed prose description of the data structures, algorithms, and design decisions of a program, right into the source code. A collection of software tools for preprocessing and reorganizing these descriptions into a legal (Pascal) notation also exists. Although we were not interested in the tools or even the syntax that Knuth suggests for literate programs, we were interested in and influenced by the idea that improved prose descriptions of the System V/MLS security policy could be embedded into the actual formal specification.

As a result of these and other influences (see [Jones 86] for instance), our own style of writing "formalized documentation" emerged. Carefully thought out prose descriptions (a la Knuth), were embedded into the formal specifications (a la Z). The resulting documentation style is one that significantly increases the likelihood of producing quality documentation.

10. Acknowledgements

We would like to thank Howard Israel, Dewayne Perry, Sera Amoroso, Ben Melamed, and our referees for their comments and suggestions.

REFERENCES

Amoroso 85 — Amoroso, S M , "A Mathematical Method for System Conceptualization," *Proceedings of the IEEE Workshop on Languages for Automation*, Palma De Mallorca, Spain, June 1985.

Bach 86 — Bach, M J., *The Design of the UNIX Operating System*, Prentice-Hall, 1986.

Bell 74 — Bell, D.E. and LaPadula, L.J., "Secure Computer Systems," Tech Rep ESD-TR-73-278, vols 1-3, MITRE Corporation, Bedford, Mass , Nov. 1973 and June 1974

Cohen 87 — Cohen, B., et al, *The Specification of Complex Systems*, Addison-Wesley, 1986.

DeMillo 79 — DeMillo, R.A., et al, "Social Processes and Proofs of Theorems and Programs," *CACM*, Vol.22, No.5, May, 1979.

DoD 85 — Department of Defense Trusted Computer System Evaluation Criteria, DoD 5200.28-STD, National Computer Security Center, Fort George Meade, Maryland, 1985.

Flink 88 — Flink, C.W. and Weiss, J D., "System V/MLS," Technical Memorandum, AT&T Bell Laboratories, 1988. (to appear in the *AT&T Technical Journal*)

Good 84 — Good, D.I., et al, Draft "Using the Gypsy Methodology," Institute for Computing Science, Univ. Texas at Austin, June 6, 1984.

Hayes 87 — Hayes, I (ed.), *Specification Case Studies*, Prentice-Hall, 1987

Hester 81 — Hester, S D , et al., "Using Documentation as a Software Design Medium," *Bell System Technical Journal*, Vol.60, No.8, October, 1981

Jones 86 — Jones, C.B., *Systematic Software Development Using VDM*, Prentice-Hall, 1986.

Kemm 80 — Kemmerer, R.A., "FDM - A Specification and Verification Methodology," in *Proc. Third Seminar on the Department of Defense Computer Security Initiative Program*, Nat. Bur. Stand., Gaithersburg, Maryland, Nov, 1980.

Kemm 86 Kemmerer, R.A., "Verification Assessment Study Final Report," Vols.I-V, C3-CR01-86, National Computer Security Center, Fort George Meade, Maryland, March, 1986.

Knuth 67 Knuth, D., "The Remaining Trouble Spots in ALGOL 60," *CACM*, Vol.10, No. 10, 1967.

Knuth 84 Knuth, D., "Literate Programming," *The Computer Journal*, Vol.27, No.2, 1984.

Naur 60 Naur, P. et al, "Report on the Algorithmic Language ALGOL 60," *CACM*, Vol.3, 1960.

Parnas 77 Parnas, D.L. and Bartussek, W., "Using Traces to Write Abstract Specifications for Software Modules," University of North Carolina Report No. TR77-012, Dec. 1977.

Perry 87 Perry, D E., "Software Interconnection Models," *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, March 30 - April 2, 1987.

Sufrin 82 Sufrin, B., "Formal Specification: Notation and Examples," in *Tools and Notations for Program Construction*, D. Neel, Ed , Cambridge Univ. Press, 1982.

Sufrin 84 Sufrin, B. and Morgan, C., "Specification of the UNIX Filing System," *IEEE-TSE*, Vol.SE-10, No.2, March, 1984.

APPENDIX: Informal Explanation of Specification Notation:

The specification framework that has been used in the examples in this paper is based on an approach called the *System Abstraction Methodology* introduced in [Amoroso 85]. A system abstraction consists of three non-empty sections: (1) a *concepts* section in which a collection of basis sets are introduced and variables are defined on these sets; (2) a description of the *initial states* in which values for all system variables are given; and (3) a collection of *events* which are essentially relations on the set of all possible states. Normal mathematical conventions are assumed and used in the writing of a specification.

Events are denoted by a possibly empty set of preconditions written above a horizontal line, and a possibly empty set of postconditions written below the same line (just as rules of inference are expressed in logic). These event descriptions are interpreted as follows: if a system state is consistent with the preconditions, then a new system state is defined that is exactly the same state, changed only to be consistent with the postconditions. For instance, the rule expressed as follows:

example1:
statement A
statement B

¬ (statement B)

is interpreted as follows: if statements A and B are true in some system state S, then there is a system state S´ that is exactly the same as state S, except that statement B is not true.

It should also be noted that objects denoted with a "tick" in a postcondition assertion, refer to the value of that object in the precondition assertions. For example, the following rule describes how one might increment an integer variable i:

increment (i).
i ∈ integer

---

i = i´ + 1

Reliability

"Reliability: Measurement, Prediction, and Application" (Slides Only)
John D. Musa, AT&T Bell Laboratories

BIOGRAPHY

JOHN D. MUSA

AT&T BELL LABORATORIES

WHIPPANY, NEW JERSEY 07981

John D. Musa is Supervisor of Software Quality at AT&T Bell Laboratories, Whippany, New Jersey. He has held a variety of software assignments in program design and has managed a number of different software projects and activities. He has also worked in the areas of computer graphics, computer security, analysis, simulation, systems engineering, and human factors engineering. His research interests include software reliability, software engineering, and software project management, and he has published over 40 papers in these and other fields. He is principal author of the pioneering book "Software Reliability: Measurement, Prediction, Application" (McGraw-Hill, 1987).

He is a member of the editorial boards of the IEEE Proceedings, IEEE Software, and Technique et Science Informatiques and a former member of the editorial board of IEEE Spectrum. He is a senior editor of the Software Engineering Institute book series.

He has been 2nd Vice President, Vice President for Technical Activities and Vice President for Publications of the IEEE Computer Society. In the latter position, he was in charge of the startup of two new magazines, the creation of another, and planning for two future magazines. He also supervised a substantial growth in book publishing operations. He has been a member of the Governing Board. Musa is past Chair of the IEEE Technical Committee on Software Engineering. He has received Computer Society Meritorious Service Awards for innovative leadership in software engineering, publications, and technical activities. He has participated in various computer conferences as Publicity Chairman, Proceedings Editor, member of the technical program committee, session chairman, and author. He is a Fellow of the IEEE, cited for "contributions to software engineering, particularly software reliability." He is a member of the ACM and a member of the Executive Committee of ACM/SIGSOFT.

MEASURING AND MANAGING SOFTWARE RELIABILITY

JOHN D.MUSA

BELL LABORATORIES

WHIPPANY, N.J. 07981

OUTLINE

1. VALUE OF SOFTWARE RELIABILITY MEASURES
2. BASIC CONCEPTS
3. EXECUTION TIME MODELS
4. APPLICATIONS (EXAMPLES)
5. EVALUATION OF USEFULNESS

WHAT IS SOFTWARE RELIABILITY?

MEASURE OF HOW WELL PROGRAM FUNCTIONS vs. OPERATIONAL REQUIREMENTS

1. CUSTOMER-ORIENTED

2. MOST IMPORTANT & MEASURABLE ASPECT OF SOFTWARE QUALITY

WHAT CAN SOFTWARE RELIABILITY MEASURES DO FOR YOU?

1. UNDERSTAND USER (YOU OR YOUR CUSTOMER) NEEDS MORE PRECISELY —
    - A. WHAT ARE *REAL* REQUIREMENTS (FORCE TRADEOFFS)?
    - B. USER: MANAGE OPERATION BETTER

2. IMPROVE RESULTS: DEVELOPMENT AND OPERATIONAL DECISIONS
    - A. SPECIFYING DESIGN GOALS, SCHEDULES, RESOURCES
    - B. MANAGING DEVELOPMENT (EVALUATE STATUS, MODEL IMPACT OF DECISIONS)
    - C. EVALUATING AND CONTROLLING QUALITY LEVEL

WHAT CAN SOFTWARE RELIABILITY MEASURES DO FOR YOU?

3. DEVELOPER: INCREASE YOUR PRODUCTIVITY
    - A. BETTER MEET CUSTOMER NEEDS
    - B. USE RESOURCES OPTIMALLY

4. HELP YOU IMPROVE POSITION OF YOUR ORGANIZATION
    - A. CUSTOMER SATISFACTION
    - B. REPUTATION
    - C. "MARKET SHARE"
    - D. "PROFITABILITY"

OUTLINE

1. VALUE OF SOFTWARE RELIABILITY MEASURES
2. BASIC CONCEPTS
3. EXECUTION TIME MODELS
4. APPLICATIONS (EXAMPLES)
5. EVALUATION OF USEFULNESS

TERMS

FAILURE: DEPARTURE OF PROGRAM OPERATION FROM REQUIREMENTS

FAULT: DEFECT CAUSING A FAILURE

ALTERNATE SOFTWARE RELIABILITY EXPRESSIONS

FAILURE INTENSITY: FAILURES/TIME PERIOD

EX: 0.025 FAILURES/HR.

SOFTWARE RELIABILITY: PROBABILITY OF FAILURE-FREE OPERATION FOR SPECIFIED TIME

EX: 0.82 FOR 8 HR. DAY

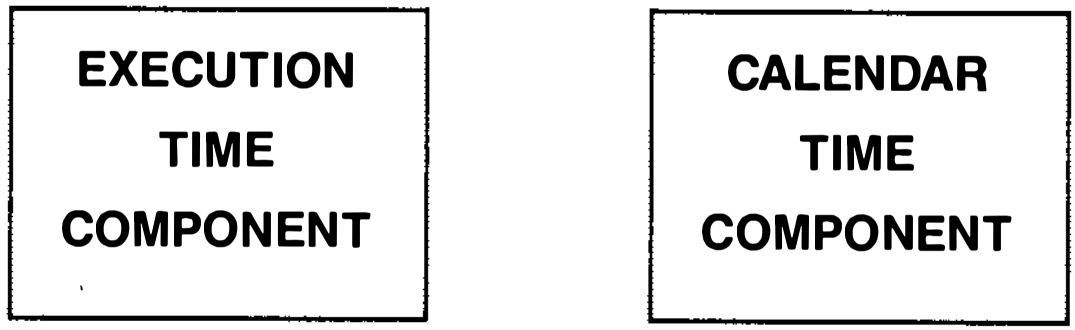
BASIC CONCEPTS-SUMMARY

SOFTWARE RELIABILITY MODELS FOCUS ON FAILURE INTENSITY VARIATION WITH TIME RESULTING FROM FAULT REMOVAL

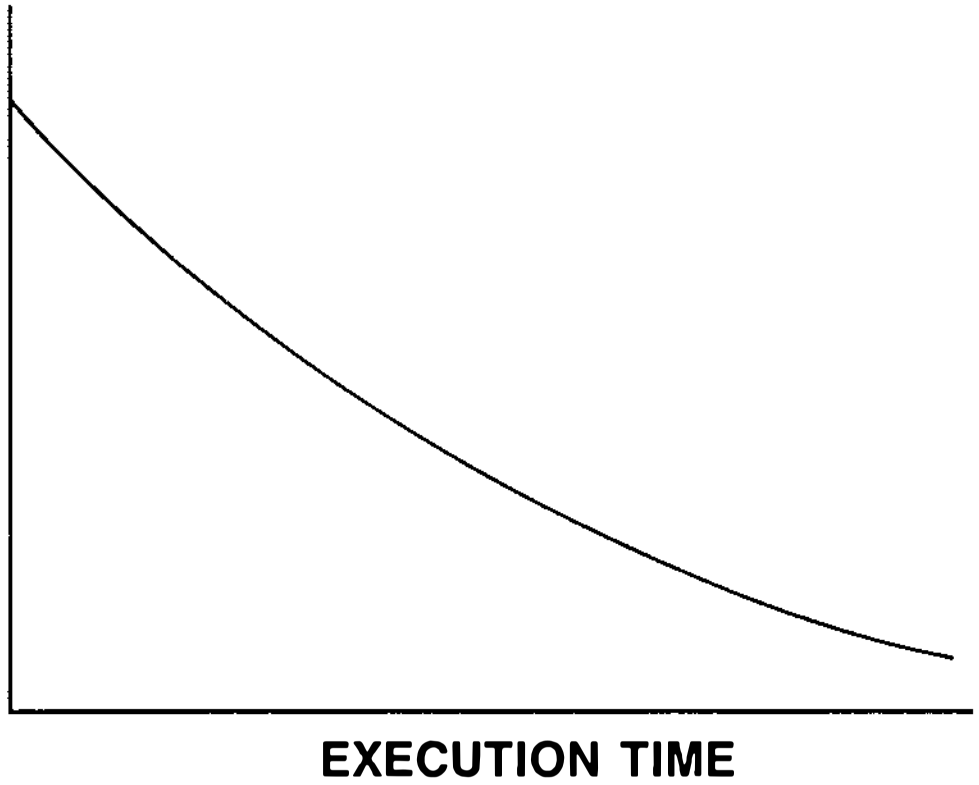
OUTLINE

1. VALUE OF SOFTWARE RELIABILITY MEASURES
2. BASIC CONCEPTS
3. EXECUTION TIME MODELS
4. APPLICATIONS (EXAMPLES)
5. EVALUATION OF USEFULNESS

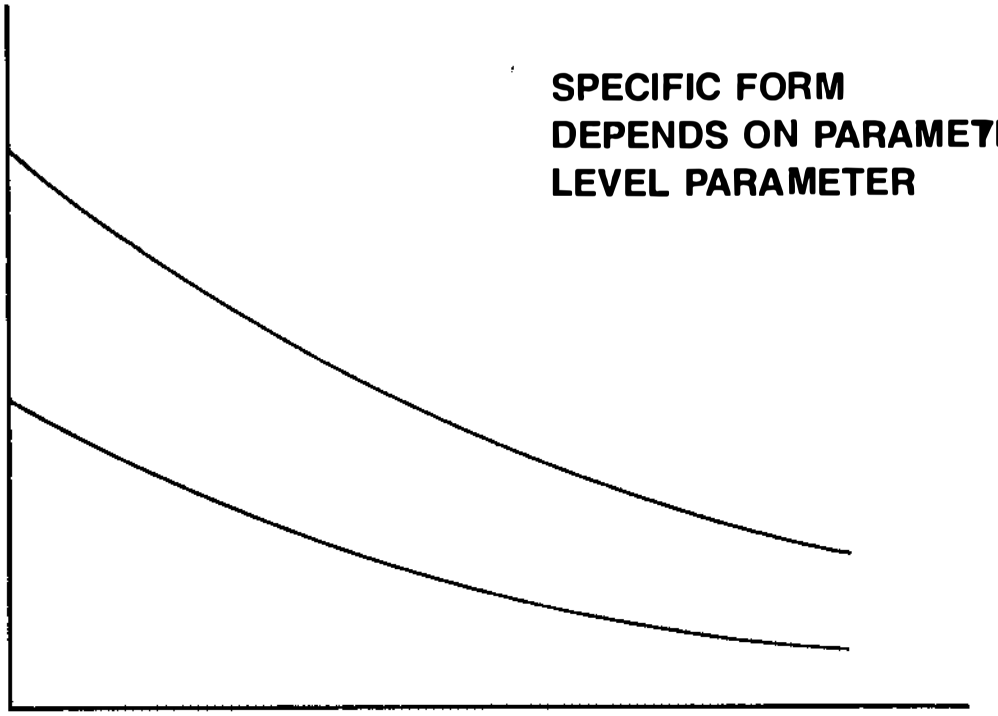
EXECUTION TIME MODELS-SCHEMATIC



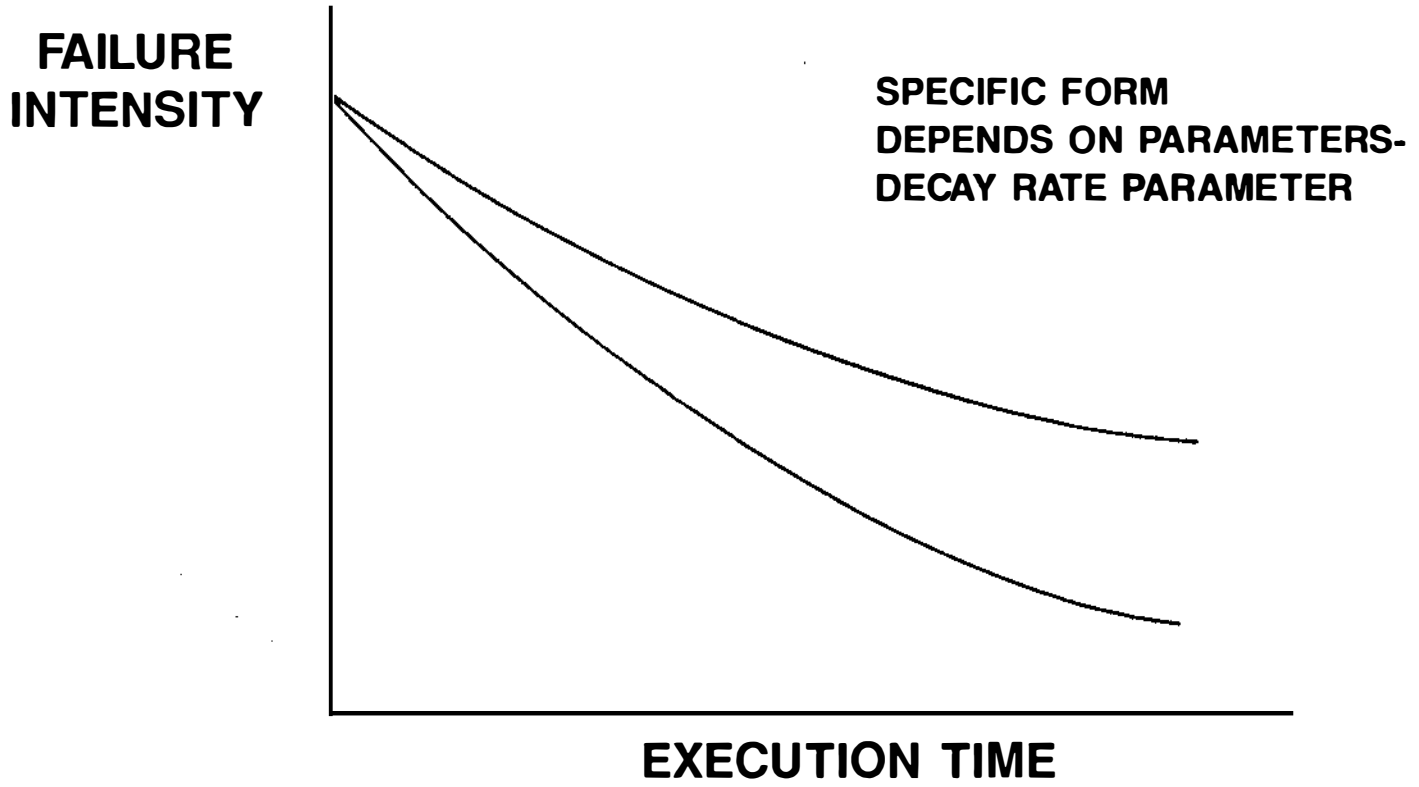
EXECUTION TIME COMPONENT-APPEARANCE



EXECUTION TIME COMPONENT



EXECUTION TIME COMPONENT

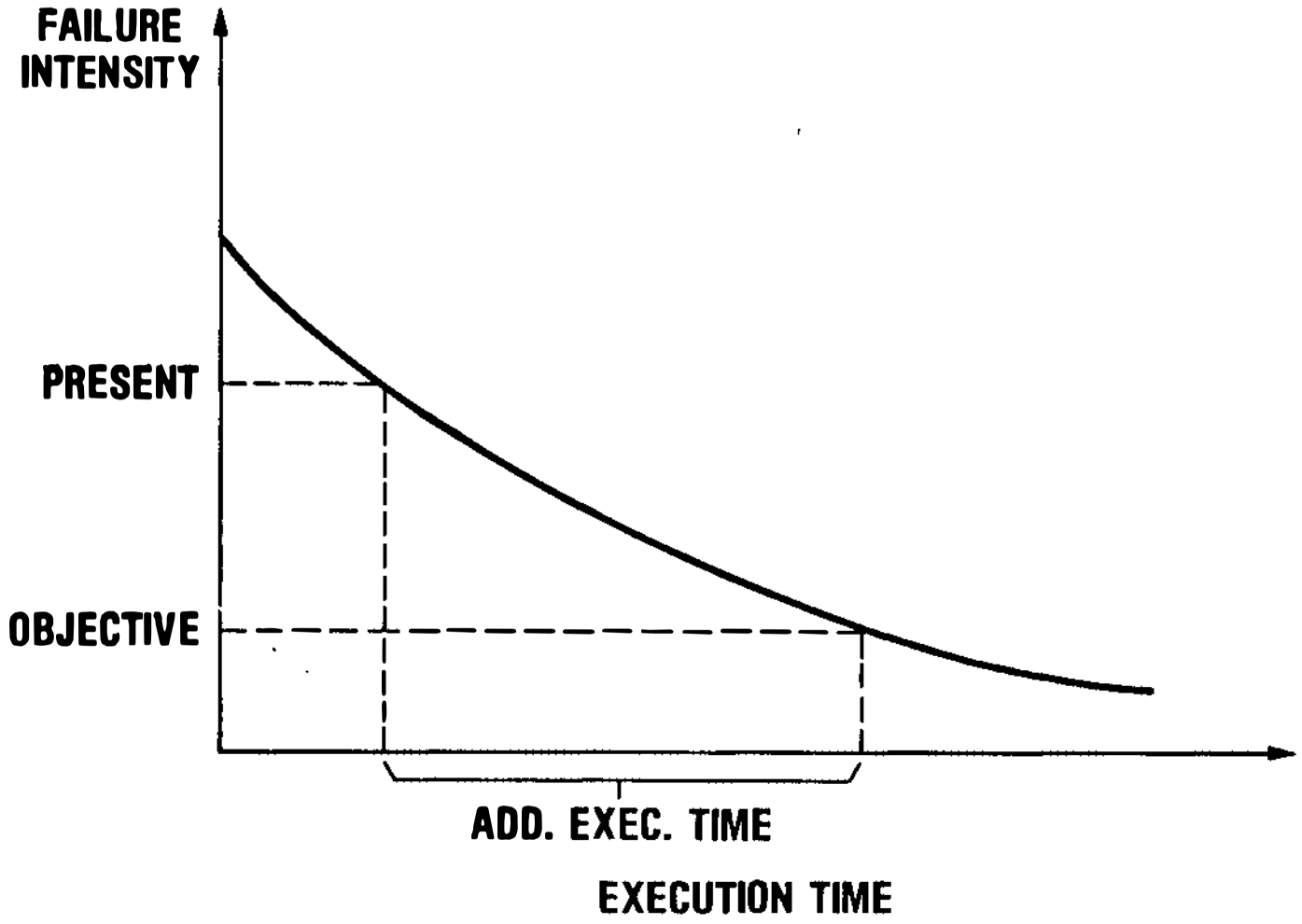


EXECUTION TIME COMPONENT-PARAMETERS

1. ESTIMATION FROM FAILURE DATA (INFERENCE)

2. PREDICTION FROM PRODUCT & DEVELOPMENT PROCESS CHARACTERISTICS

EXECUTION TIME COMPONENT-PREDICTIONS



CALENDAR TIME COMPONENT

DETERMINE CALENDAR TIME TO EXECUTION TIME RATIO, BASED ON RESOURCE LIMITATIONS THAT AFFECT PACE OF OPERATION

OUTLINE

1. VALUE OF SOFTWARE RELIABILITY MEASURES
2. BASIC CONCEPTS
3. EXECUTION TIME MODELS
4. APPLICATIONS (EXAMPLES)
5. EVALUATION OF USEFULNESS

SOFTWARE RELIABILITY MEASUREMENT APPLICATIONS (EXAMPLES)

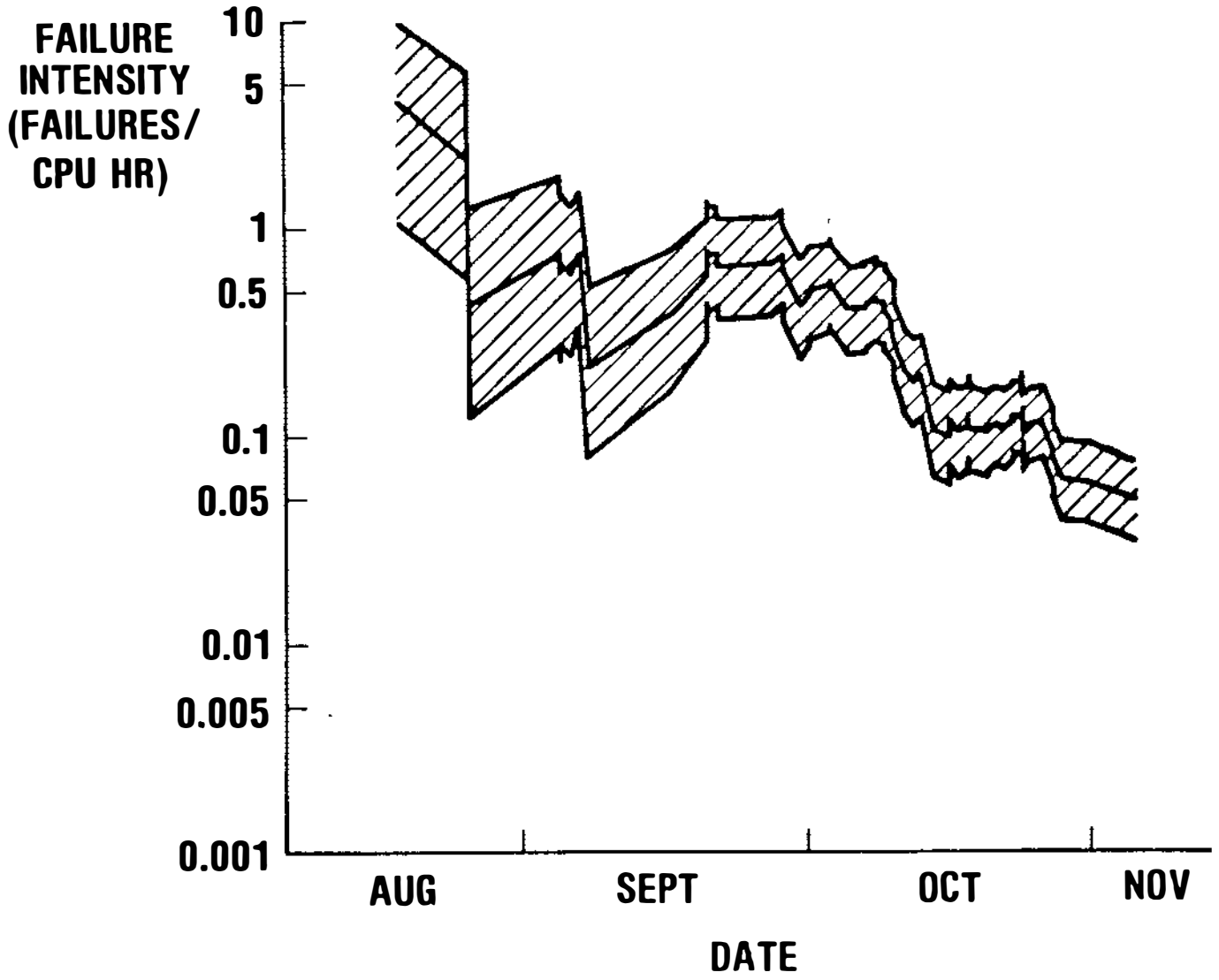
1. SPECIFY
    - A. TRADEOFF AND SELECT DESIGN GOALS
    - B. PLAN RESOURCES, COSTS, SCHEDULES
2. CONTROL
    - A. MONITOR AND PREDICT DEVELOPMENT PROGRESS
    - B. CONTROL CHANGE-OPERATIONAL SOFTWARE
3. EVALUATE
    - A. MEASURE ACHIEVED SOFTWARE QUALITY
    - B. DETERMINE EFFECT OF SOFTWARE ENGINEERING TECHNOLOGY

SOFTWARE RELIABILITY PREDICTION BASIC MODEL
SYSTEM T1

BASED ON SAMPLE OF 136 TEST FAILURES
EXECUTION TIME IS 25.34 HRS
FAILURE INTENSITY OBJECTIVE IS .36e-01 PER HOUR
CALENDAR TIME TO DATE IS 96 DAYS
PRESENT DATE: 11/9/73

	CONF. LIMITS				MOST		CONF. LIMITS		
	95%	90%	75%	50%	LIKELY	50	75%	90%	95%
TOTAL FAILURES	139.	139.	140.	141.	142	144.	145.	147.	149
FAILURE INTENSITIES (1/1000* HRS)									
INITIAL F.I.	14390	14920	15766	16593	17785	19000	19869	20782	21368
PRESENT F.I.	432.5	472.6	[542.7]	618.9	[743.5]	890.2	[1008]	1146	1242
*** ADDITIONAL REQUIREMENTS TO MOST FAILURE INTENSITY OBJECTIVE ***									
FAILURES	3	3	4	4	6	7	9	11	12
EXEC. TIME (HR)	16.2	17.3	19.1	21.1	24.2	27.8	30.7	34.2	36.7
CAL. TIME (DAYS)	3.81	4.11	4.62	5.18	6.16	7.41	8.51	9.90	10.9
COMPLETION DATE	111573	111673	[111673]	111973	[112073]	112173	[112273]	112373	112673

MONITORING TEST PROGRESS



OUTLINE

1. VALUE OF SOFTWARE RELIABILITY MEASURES
2. BASIC CONCEPTS
3. EXECUTION TIME MODELS
4. APPLICATIONS (EXAMPLES)
5. EVALUATION OF USEFULNESS

EVALUATION OF USEFULNESS

1. HOW WELL DOES IT WORK?

2. WHAT DOES IT COST?

3. HOW CAN I LEARN MORE?

HOW WELL DOES IT WORK?: EXAMPLES

1. HEWLETT PACKARD: TERMINAL FIRMWARE
   - THEY REPORT 12% ACCURACY IN PREDICTING FIELD FAILURE RATE

2. AT&T: UNIX® SOFTWARE DEVELOPMENT ENVIRONMENT
   - ESTIMATED END OF TEST WITHIN 1 WEEK FOR LAST 80% OF SYSTEM TEST PERIOD

3. AT&T: 5 ESS TELECOMMUNICATIONS SWITCHING SYSTEM
   - PREDICTED FIELD FAILURES OVER 2 YR FOR 3 RELEASES WITH 5-10% ACCURACY

WHAT DOES IT COST?

1. ABOUT 0.1 - 0.2% OF PROJECT

2. BENEFIT TYPICALLY ORDER OF MAGNITUDE GREATER

   EX: 2 YR. PROJECT, 6 MO. SYSTEM TEST, 4% SAVING

HOW CAN I LEARN MORE?

BOOK - "SOFTWARE RELIABILITY: MEASUREMENT, PREDICTION, APPLICATION" BY MUSA, IANNINO, OKUMOTO - McGRAW-HILL, 1987

Notes

Notes

Notes

1988 PROCEEDINGS ORDER FORM

Pacific Northwest Software Quality Conference

To order a copy of the 1988 Proceedings, please send a check in the amount of $30.00 to:

PNSQC
c/o Lawrence & Craig, Inc.
P.O. Box 40244
Portland, OR 97240

Name _____

Title _____

Affiliation _____

Mailing Address _____

City, State _____ Zip _____

Daytime Telephone _____