EIGHTH ANNUAL PACIFIC NORTHWEST

SOFTWARE QUALITY CONFERENCE

-

October 29 - 31, 1990

Oregon Convention Center

Portland, Oregon

TABLE OF CONTENTS

Tools Track Day One

Tools Track Day Two

Engineering Track Day One

Preface

Sue Bartlett

Welcome to the Eighth Annual Pacific Northwest Software Quality Conference. We are pleased to publish these Proceedings which contain the papers presented at the two day technical program. Out of all the abstracts submitted for review, 26 were selected to represent the state of the art in Software Quality. An additional nine invited speakers representing experts in the industry have also been included.

The abstracts were reviewed by the Program Committee. They met to consider each abstract, discuss its merits and make the final decision. Once picked, each paper was reviewed by three people and comments were relayed back to the author. Every effort was made to make the review process as fair as possible and to select the best papers for presentation in this Proceedings. We feel that the papers selected are of the highest quality and report important research results.

A significant change this year was going to two days which allowed us to give you the longer talks you have requested and enabled the addition of invited experts to supplement the program.

I thank Mark Johnson and Elicia Harrell, the Program Committee co-chairs, for all their hard work at coordination and doing what it takes to pull the program together. Many thanks also to the members of the Program Committee for their time and energy without which this whole process would not work.

I also want to thank the rest of the members of the full committee, whose names are listed in the next section, who put such hard work in and attended endless meetings to make this conference happen.

Finally I want to give special thanks to Terri Moore at Pacific Agenda for keeping us all on track and who expertly handled the organizational and administrative tasks associated with the conference in addition to putting together this Proceedings.

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Sue Bartlett - President and Chair
Mentor Graphics

Paul Blattner - Vice President; Publicity Chair
Quality Software Engineering

Douglas R. Saxton - Secretary
Portland General Electric

Steve Shellans - Treasurer; Keynote;
Software Excellance Award
Tektronix, Inc.

Mark Johnson - Technical Program Co-Chair
Mentor Graphics

Elicia Harrell - Technical Program Co-Chair
Intel Corporation

Bill Sundermeier - Exhibits
Cadre Technologies

Sandhi Bhide - Workshops
Mentor Graphics

CONFERENCE PLANNING COMITTEE

Hilly Alexander
ADP Dealer Services

Lowell Billings
Infotec Development, Inc.

Greg Boone
CASE Research

John Burley
Victory Software Systems

Thomas Coleman
Timberline Software

Margie Davis
ADP Dealer Services

Dave Dickmann
Hewlett Packard

Kathy Fieldstad
II Morrow

Cynthia Gens
Portland State University

Michael Green
Software Professionals, Inc.

Dick Hamlet
Portland State University

Warren Harrison
Portland State University

Bryan Hilterbrand
Tektronix, Inc.

Dan Hoffman
University of Victoria

Bill Junk
University of Idaho

Debra Lee
Intel Corporation

Ray Lischner
Mentor Graphics

Ken Maddox
Software Association of Oregon

Chuck Martiny
Tektronix, Inc.

Shannon Nelson
Intel Corporation

Misty Pesek
Servio Corporation

Lisa Powell
Flothe, Johnson and Associates

William Smith
Standard Insurance

Lee Thomas
A T & E

George Tice
Mentor Graphics

Karen Ward
Portland General Electric

Dan Whitiker
Evergreen Technologies

Nancy Winston
Mentor Graphics

PRESENTERS

Andrew Babbitt
Shari Tims Powell
PORTLAND STATE UNIVERSITY
Department of Computer Science
P.O. Box 751
Portland, OR 97207

Dori Barnes
J. Malsbury
PRINCETON PLASMA PHYSICS LAB
P.O. Box 451
Princeton, NJ 08543

Victor R. Basili
UNIVERSITY OF MARYLAND
Computer Science Department
College Park, MD 20742

Boris Beizer
1232 Glenbrook Road
Huntingdon Valley, PA 19006

Ron Blair
Dan Uhrich
HONEYWELL, INC.
SSDC MN63-C040
1000 Boone Avenue N
Golden Valley, MN 55427

Frank A. Cioch
Fatma Mili
OAKLAND UNIVERSITY
Department of Computer Science/Eng.
Rochester, MI 48309

James S. Collofello
Bakul P. Gosalia
ARIZONA STATE UNIVERSITY
Computer Science Department
Tempe, AZ 85287

Amitava Datta
Prabhaker Mateti
WRIGHT STATE UNIVERSITY
Department of Computer Science/Eng.
Dayton, OH 45435

Michael Deutsch
HUGHES AIRCRAFT COMPANY
Building 618 MS B204
P.O. Box 3310
Fullerton, CA 92634

Terry L. Downs
Tai-Ya Lee
AT & T BELL LABS
Rm. 1J-321
480 Red Hill Road
Middleton, NJ 07748

Phyllis G. Frankl
Roong-Ko Doong
POLYTECHNIC UNIVERSITY
Department of Computer Science
333 Jay Street
Brooklyn, NY 11201

Robert Glass
Box 213
State College, PA 16804

Warren Harrison
PORTLAND STATE UNIVERSITY
Department of Computer Science
P.O. Box 751
Portland, OR 97297

Daniel Hoffman
UNIVERSITY OF VICTORIA
Department of Computer Science
P.O. Box 1700
Victoria, BC V8W 2Y2

Dennis Kafura
Ashok Yerneni
VIRGINIA TECH
Department of Computer Science
636 McBryde Hall
Blacksburg, VA 24061-0106

James Kirby
Robert Chi Tau Lai
David Weiss
SW PRODUCTIVITY CONSORTIUM
SPC Building
2214 Rock Hill Road
Herndon, VA 22070

Keith Koplitz
MENTOR GRAPHICS
8500 S.W. Creekside Place
Beaverton, OR 97005-7191

Norm Kerth
Robert Rhodes
John Burley
ELITE SYSTEMS
7400 S.W. Barnes Rd. #911
Portland, OR 97229

James K. Larson
TEKTRONIX, INC.
P.O. Box 500
Beaverton, OR 97007

Daniel B. Leifker
THE MITRE CORPORATION
1120 NASA Road One
Houston, TX 77058

Diane Mularz
Roberta Hutchison
Douglas Cook
William Evanco
Francis Maginnis
THE MITRE CORPORATION
7525 Colshire Dr.
MS Z645
McLean, VA 22102

Hausi Müller
UNIVERISTY OF VICTORIA
Department of Computer Science
P.O. Box 1700
Victoria, BC V8W 2Y2

Thomas Ostrand
SIEMENS RESEARCH LABORATORIES
105 College Road East
Princeton, NJ 08540

Regina Palmer
MARTIN MARLETTA SPACE SYS CO
P.O. Box 179
M/S H4330
Denver, CO 80201

Shari Pfleeger
Joseph C. Fitzgerald, Jr.
CONTEL TECHNOLOGY CENTER
15000 Conference Center Drive
P.O. Box 10814
Chantilly, VA 22021-3808

James M. Purtilo
Elizabeth L. White
UNIVERSITY OF MARYLAND
Computer Science Department
College Park, MD 20742

Debra Richardson
UNIVERSITY OF CALIFORNIA, IRVINE
Department of Computer Science
Irvine, CA 92717

William Roetzheim
13518 Jamul Drive
Jamul, CA 92035

M.H. Samadzadeh
Sekaran Nanja
OKLAHOMA STATE UNIVERSITY
Department of Computer Science
219 Math Sciences
Stillwater, OK 74078-0599

Kathrine Stevens
ADP
Dealer Services Division
2525 S.W. First St., Suite 450
Portland, OR 97201

Ilkka Tervonen
UNIVERSITY OF OULU
Department of Info Processing Sc
SF-90570
Oulu, Finland

Jeffrey Voas
Larry Morell
COLLEGE OF WILLIAM AND MARY
Department of Computer Science
Williamsburg, VA 23185

Mark Weiser
XEROX PALO ALTO RESEARCH CENTER
Computer Science Laboratory
333 Coyote Hill Road
Palo Alto, CA 94304

Elaine Weyuker
COURANT INSTITUTE OF MATH
Department of Computer Science
251 Mercer Street
New York, NY 10012

Mark Whiting
PACIFIC NORTHWEST LABORATORY
P.O. Box 999
Richland, WA 99352

EXHIBITORS

Bill Sundermeier
CADRE TECHNOLOGIES
19545 NW Van Neuman #200
Beaverton, OR 97006
503/690-1318

Debbie Boren
CASE RESEARCH CORPORATION
155 108th Avenue NE, #210
Bellevue, WA 98004
206/453-9900

Paul White
COMP VIEW, INC.
13405 NW Cornell Rd.
Portland, OR 97229
503/641-8439

Keith Pettingill
DIGITAL EQUIPMENT CORPORATION
7995 SW Mohawk Drive
Tualatin, OR 97062
503/691-3227

Norman Kerth
ELITE SYSTEMS
7400 SW Barnes Rd., #911
Portland, OR 97225
503/297-8677

Jane Neall
KNOWLEDGEWARE, INC.
3340 Peachtree Road
Atlanta, GA 30326
404/231-8575

David Fanning
KOLISCH, HARTWELL & DICKINSON
520 S.W. Yamhill, Suite 200
Portland, OR 97204
503/224-6655

Ned Hammond
LOGIC GENERAL, INC.
7905 SW Nimbus Ave.
Beaverton, OR 97005
503/643-8210

Sally Gilbert
NICHOLSTONE COMPANIES
3203 NW Guam
Portland, OR 97210
503/228-6113

Shawn Wall
POWELL'S TECHNICAL BOOKSTORE
32 NW Eleventh Street
Portland, OR 97209
503/228-3906

Paul Blattner
QUALITY SOFTWARE ENGINEERING
P.O. Box 303
Beaverton, OR 97075
503/538-8256

Craig Ralston
SAGE SOFTWARE
1700 NW 167th Place
Beaverton, OR 97006
503/645-1150

Teresa Harrison
SET LABORATORIES, INC.
P.O. Box 83627
Portland, OR 97283
503/289-4758

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

Richard Savage
SOFTWARE SECURITY
4660 Paradise Dr.
Tiburon, CA 94920
415/435-6261

Denise Hartvickson
SUN MICROSYSTEMS, INC.
9900 SW Greenburg Rd., #240
Portland, OR 97223
503/684-9001

Randall Kimball
TECHNICAL SOLUTIONS, INC.
16199 N.W. Joscelyn St.
Beaverton, OR 97006
503/591-8860

David Ouellette
TERRA PACIFIC WRITING CORP.
P.O. Box 1244
Corvallis, OR 97339
503/754-6043

VAN NOSTRUM REINHOLD
c/o Boris Beizer
1232 Glenbrook Rd.
Huntingdon Valley, PA 19006
215/572-5580

The Experience Factory: Packaging Software Experience

Victor R. Basili

Institute for Advanced Computer Studies

Department of Computer Science

University of Maryland

ABSTRACT

This talk offers a paradigm for software development that treats it as an experimental activity. It is based on the concept that software development should be a closed loop process. It provides built-in mechanisms for learning how to develop software better and reusing previous experience in the forms of knowledge, processes and products. It uses models and measures to aid in the tasks of characterization, evaluation and motivation. The approach has evolved based on work performed at the Software Engineering Laboratory at NASA/GSFC and other organizations. It offers an evolutionary improvement paradigm tailored to the software business, the Quality Improvement Paradigm and a mechanism for establishing project and corporate goals and measuring against those goals, the Goal/Quality/Metric Paradigm. It proposes an organizational scheme that separates the project-specific focus from the organization's learning and reuse focuses of software development, supporting the packaging of software competencies and supplying them back to projects, the Experience Factory.

Business Requirements

Any successful business requires:

A combination of technical and managerial solutions

A well defined set of product needs
- to satisfy the customer
- to assist the developer in accomplishing those needs
- to create competencies for future business

A well defined set of processes
- to accomplish what needs to be accomplished
- to control development
- to improve the business

A closed loop process that supports learning and feedback

Key technologies for supporting these needs include:
- modeling
- measurement
- reuse

of processes, products, and other forms of knowledge relevant to our business

Evolution of Concepts

Evolving concepts for over 14 years
- In the Software Engineering Laboratory (SEL) at NASA/GSFC
- Other Industrial Environments

Phase I: Understanding

Worked on understanding what we could about the environment and measurement
- measured what we could
- used available models
- built baselines and models
- developed the Goal/Question/Metric (GQM) Paradigm

What we learned about

Understanding

There are factors that create similarities and differences among projects
- *one model does not work in all situations*

There is a direct relationship between process and product
- *we must choose the right processes to create the desired product characteristics*

Measurement is necessary and must be based on the appropriate goals and models
- *appropriate measurement provides visibility*

Evaluation and feedback are necessary for project control
- *we need a closed loop process for project control*

Evolution of Concepts

Phase II: Continuous Improvement

Worked on improving the process and product
- evaluated and fed back information to the project
- experimented with technologies
- developed the Quality Improvement Paradigm (QIP)
- evolved the Goal/Question/Metric Paradigm
- began formalizing process, product, knowledge and quality models

What we learned about

Continuous Improvement

Software development follows an experimental paradigm
- *learning and feedback are natural activities for software development and maintenance*

Process, product, knowledge, and quality models need to be better defined and tailored
- *we need evolving definitions of the components of the software business*

Evaluation and feedback are necessary for learning
- *we need a closed loop for long range improvement*

New technologies must be continually introduced
- *we need to experiment with technologies*

Reusing experience in the form of processes, products, and other forms of knowledge is essential for improvement
- *reuse of knowledge is the basis of improvement*

Evolution of Concepts

Phase III: Packaging Experiences for Reuse

We have been packaging experiences for reuse
- choosing potentially reusable experiences
- studying notations for defining reusable experiences
- defining models of tailorable reusable experiences
- defining process models for reusing experience
- developed the experience factory concept and integrated it with an evolved QIP and GQM

What we learned about

Packaging Experiences for Reuse

Experience needs to be packaged
- *we must build competencies in software*

Experiences must be evaluated for reuse potential
- *an analysis processes is required*

Software development and maintenance processes must support reuse of experience
- *we must say how and when to reuse*

A variety of experiences can be packaged
- *we can build process, product, resource, defect and quality models*

Experiences can be packaged in a variety of ways
- *we can use equations, histograms, algorithms*

Packaged experiences need to be integrated
- *we need an experience base of integrated information*

Overview of the Presentation

This talk offers:

an evolutionary improvement paradigm tailored for the software business

Quality Improvement Paradigm

a paradigm for establishing project and corporate goals and a mechanism for measuring against those goals

Goal/Question/Metric Paradigm

an organizational approach for building software competencies and supplying them to projects

Experience Factory

Quality Improvement Paradigm

Characterize the current project and its environment.

Set the quantifiable **goals** for successful project performance and improvement.

Choose the appropriate **process** model and supporting methods and tools for this project.

Execute the **processes**, construct the products, collect and validate the prescribed data ,and **analyze** it to provide real-time feedback for corrective action.

Analyze the **data** to evaluate the current practices, determine problems, record findings, and make recommendations for improvement for future project improvements.

Package the experience in the form of updated and refined models and other forms of structured knowledge gained from this and prior projects and save it in an experience base so it is available for future projects.

Characterizing the Environment

Characterizing the Environment

Classify the current project with respect to a variety of characteristics

Find the class of projects with similar characteristics and goals to the project being developed

Distinguish the relevant project environment for the current project

Provides a context for
- Goal Definition
- Reusable experience/Objects
- Process Selection
- Evaluation/Comparison
- Prediction

Project Characteristics
Environmental Factors

People Factors:
number of people, level of expertise, group organization, problem experience, process experience,...

Problem Factors:
application domain, newness to state of the art, susceptibility to change, problem constraints, ...

Process Factors:
life cycle model, methods, techniques, tools, programming language, other notations, ...

Product Factors:
deliverables, system size, required qualities, e.g., reliability, portability, ...

Resource Factors:
target and development machines, calendar time, budget, existing software, ...

Setting the Goals

Setting the Goals

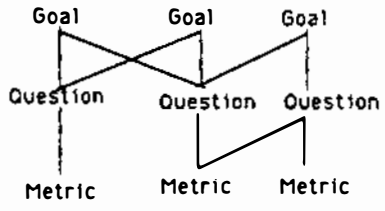
Need to establish goals for the processes and products

Goals should be measurable, driven by the models

Goals should be defined from a variety of perspectives:
- User,
- Customer
- Project,
- Corporation

There are a variety of mechanisms for defining measurable goals:
- Quality Function Deployment Approach (QFD)
- Goal/Question/Metric Paradigm (GQM)
- Software Quality Metrics Approach (SQM)

GQM



A Goal links two models:

a model of the object of interest

a model of the quality perspective

and develops an integrated model

Example

Goal: Characterize the final product with respect to defect classes

Question: What is the error distribution by phase of entry

Metric: Requirements Errors

OVERVIEW OF THE GQM APPROACH

Develop a set of corporate, division and project goals for productivity and quality, e.g., customer satisfaction, on time delivery, improved quality

Generate questions (based upon models) that define those goals as completely as possible in a quantifiable way.

Specify the measures needed to be collected to answer those questions and to track process and product conformance to the goals.

Develop mechanisms for data collection.

Collect, validate and analyze the data in real time to provide feedback to projects for corrective action.

Analyze the data in a post mortem fashion to assess conformance to the goals and make recommendations for future improvements.

Goal Generation Template

Goals may be defined for any object, for a variety of reasons, with respect to various models of quality, form various points of view, relative to a particular environment.

Purpose:
Analyze some
(objects: processes, products, other experience models)
for the purpose of
(why: characterization, evaluation, prediction, motivation, improvement)

Perspective:
with respect to
(focus: cost, correctness, defects, changes, reliability, user friendliness,...)
from the point of view of
(who: user, customer, manager, developer, corporation,...)

Environment:
in the following context
(problem factors, people factors, resource factors, process factors,...)

Process Goal Question Guidelines

Process Conformance:

Characterize the process quantitatively and assess how well the process is performed.

Domain Conformance:

Characterize the object of the process and evaluate the knowledge of the object and its domain by the process performers.

Focus:

Analyze the output of the process according to some quality model and some viewpoint.

Feedback:

What has been learned about the process, its application, the product domain, or any other process or product?

Process Goal Example

Analyze the system test process for the purpose of evaluation with respect to defect slippage from the point of view of the corporation.

System Test Process Model:

Goal: Generate a set of tests consistent with the complexity and importance of each requirement.

Process: (1) Enumerate the requirements, (2) Rate importance by marketing, (3) Rate complexity by system tester, (4) ...

Defect Slippage Model:

Let **Fc** = the ratio of faults found in system test to the faults found after system test on this project.

Let **Fs** = the ratio of faults found in system test to the faults found after system test in the set of projects used as a basis forcomparison.

Let **QF = Fc/Fs** = the relationship of system test on this project to faults as compared to the average the appropriate basis set.

Simple Interpretation of Defect Slippage Model

if QF
  > 1 then
    method good
    check process conformance
    if process conformance poor
      improve process or process conformance
    check domain conformance
    if domain conformance poor
      improve object or domain training

  = 1 then
    method average
    if cost lower than normal
      method cost effective
      check process conformance
    ...

  < 1 then
    check process conformance
    if process conformance good
      check domain conformance
      if domain conformance good
        method poor for this class of project

Choosing the Execution Model

Choosing the Execution Model

Choose an appropriate generic process model
  Define its goal and give its definition

Choose and tailor an appropriate and integrated set of methods
  Define their goals and give their definitions

Choose and tailor an appropriate and integrated set of techniques
  Define their goals and give their definitions

Note: Choosing and tailoring are always done in the context of the environment, project characteristics, and goals established for the products and other processes

DEFINING PROCESS TERMINOLOGY

How do we differentiate between a technique, method, life cycle model, and engineering?

TECHNIQUE: A technology for constructing or assessing the software, e.g., reading.

METHOD: An organized approach based upon applying some technique, e.g., design inspections.

LIFE CYCLE MODEL: An integrated set of methods that covers the life cycle, e.g., an incremental development model using structured design, design inspections, etc.

ENGINEERING: The application and tailoring of techniques, methods and process models to the problem, project and organization.

LIFE CYCLE MODELS

Waterfall model

Sequential

Good when

Problem and solution well understood

Iterative enhancement model

Incremental versions developed

Good when

Problem or solution not well understood

Schedule for full function a risk

Requirements changing

Prototyping model

Experimental version

Good when

User unsure of needs

Some aspect of the system unclear

Experimental analysis needed

TECHNIQUE: Reading

Input object: Requirements, specification, design, code, test plan, ...

Output object: set of anomalies

Approach: Sequential, path analysis, stepwise abstraction, ...

Formality: Reading, correctness demonstrations, ...

Emphasis: Fault detection, traceability, performance, ...

Method: Walk-throughs, inspections, reviews, ...

Consumers: User, designer, tester, maintainer, ...

Product qualities: Correctness, reliability, efficiency, portability, ...

Process qualities: Adherence to method, integration into process model, ...

Quality view: Assurance, control, ...

Process Needs

There is a software technology, but it is not simple

We need flexible definitions and goals for processes

We need experimentation and analysis of the various technologies to understand
their strengths and weaknesses
when and where they are appropriate
how to tailor them to a specific project

We need education and training in the
application
life cycle models
methods
techniques
tools
e.g. what training is done in reading?

DEFINING PROCESS GOALS

What are the goals of the Software Development Process?

From the project management perspective:

Develop a set of documents that all represent the same system

From the user's perspective:

Develop a system that satisfies the user's needs with respect to functionality, quality, cost, etc.

From the corporate perspective:

Improve the organization's ability to develop quality systems productively and profitably

DEFINING PROCESS GOALS

Goals help define the life cycle model, methods, and techniques

What are the goals of the test activity?

To assess quality or to find failures?

Selection of activities depends upon the answer

Assess quality:
- Tests based upon user operational scenario
- Statistically based testing technique
- Reliability modeling for assessment

Find failures:
- Test a function at a time
- General to specific
- Reliability models not appropriate

Executing the Processes

Executing the Processes

Access packaged experience

Some types of analysis done in close to real time

Feedback for corrective action

Data collection must be integrated into the processes, not an add on

Data validation important

Automated support necessary to:
- support mechanical tasks
- deal with large amounts of data and information needed for analysis

Kinds of Data Collected

Resource Data:
- Effort by activity, phase, type of personnel
- Computer time
- Calendar time

Change/Defect Data:
- Changes and defects by various classification schemes

Process Data:
- Process definition
- Process conformance
- Domain understanding

Product Data:
- Product characteristics
  - logical, e.g., application domain, function
  - physical, e.g. size, structure
- Use and context information

Analyzing and Packaging Experience

Analyzing the Data

Based upon the goals, we interpret the data that has been collected.

We can use this data to:

characterize and understand, e.g.,
- what project characteristics effect the choice of processes, methods and techniques
- which phase is typically the greatest source of errors

evaluate and analyze, e.g.
- what is the statement coverage of the acceptance test plan?
- does the Cleanroom Process reduce the rework effort?

predict and control, e.g.,
- given a set of project characteristics, what is the expected cost and reliability, based upon our history

motivate and improve, e.g.,
- for what classes of errors is a particular technique most effective

Packaging the Experience

Improving the software process and product requires the
- continual accumulation of evaluated experiences (learning)
- in a form that can be effectively understood and modified (**experience models**)
- into a repository of integrated experience models (**experience base**)
- that can be accessed and modified to meet the needs of the current project (**reuse**)

Systematic learning requires support for
- recording experience
- off-line generalizing and tailoring of experience
- formalizing experience

Packaging useful experience requires
- a variety of models and formal notations that are tailorable, extendible, understandable, flexible and accessible

Packaging the Experience

An effective experience base must contain
- accessible and integrated set of analyzed, synthesized, and packaged experience models that captures the local experiences

Systematic reuse requires support for
- using existing experience
- on-line generalizing or tailoring of candidate experience

This combination of ingredients requires an organizational structure that supports them.

This includes:
- a software evolution model that supports reuse
- a set of processes for learning, packaging, and storing experience
- the integration of these two functions

What kinds of experience can we package?

Resource Baselines and Models

Change and Defect Baselines and Models

Product Baselines and Models

Process Definitions and Models

Method and Technique Models and Evaluations

Products

Lessons Learned

Quality Models

Forms of Packaged Experience

Equations defining the relationship between variables,

e.g. Effort = $a*Size^b$

Histograms or pie charts of raw or analyzed data
e.g. % of each class of fault

Graphs defining ranges of "normal"
e.g. graphs of size growth over time with confidence levels

Specific lessons learned
associated with project types, phases, activities
e.g. reading by stepwise refinement is most effective for finding interface faults
in the form of risks or recommendations
e.g. definition of a unit for unit test in Ada needs to be carefully defined

models or algorithms specifying the processes, methods, or techniques
e.g. an SADT diagram defining Design Inspections with the reading technique a variable dependent upon the focus and reader perspective

Why has reuse been a problem?

Need to reuse more than code

Reuse of experience has been too informal

Reuse has not been fully incorporated into the development or maintenance process models

Experience needs to be analyzed for its reuse potential

Experience needs to be tailored

Experience needs to be packaged

Project focus is delivery, not reuse

The Experience Factory

Logical and/or physical organization that supports project developments by

analyzing and synthesizing all kinds of experience

acting as a repository for such experience

supplying that experience to various projects on demand

It packages experience by building

informal, formal or schematized, and productized models and measures

of various software processes, products, and other forms of knowledge

via people, documents, and automated support

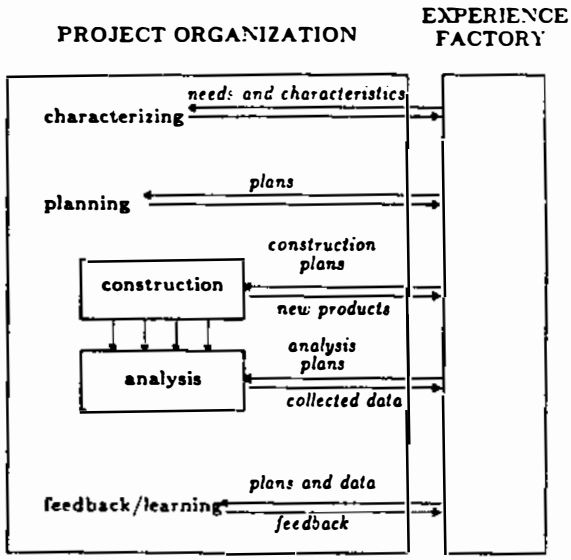
The Experience Factory

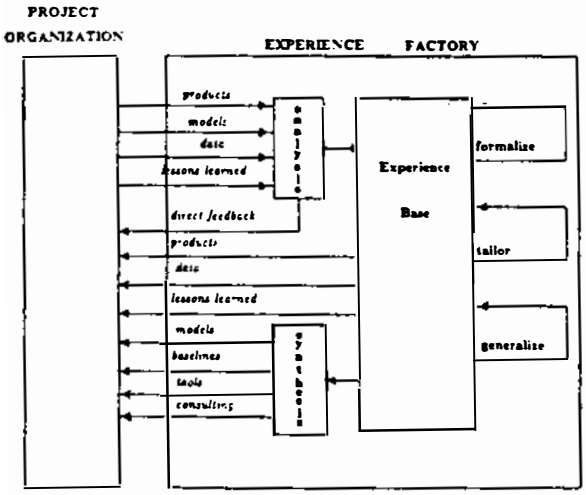
Requires separate organizations

- Project Organization
- Experience Factory

WHY?

- Different focuses/priorities
- Different process models
- Different expertise requirements





What kinds of experience can we package?

Resource Baselines/Models

Example:

Cost Models

Resource allocation models

- staffing
- schedule
- computer utilization

The relationship between resources and various factors,

e.g. specific methods
customer complexity
the application
the environment
defect classes

What kinds of experience can we package?

Change and Defect Baselines and Models

Examples:

- Change baselines by various classifications
- Defect baselines by various classifications
- Defect prediction models
- Reliability Models

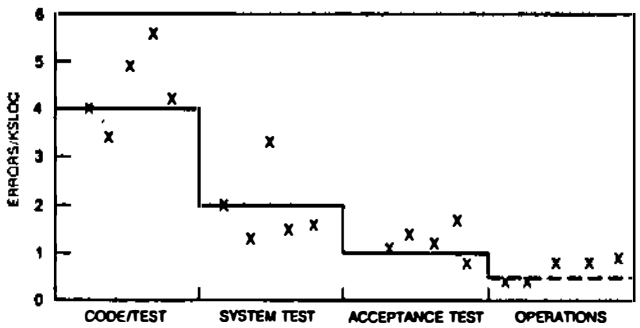
PLANNING AIDS FOR ONE ENVIRONMENT*

ESTIMATED PROJECT COST	EFFORT = 1.48 *KSLOC $^{.98}$	
EFFORT DISTRIBUTION	PRELIMINARY DESIGN	15%
	DETAILED DESIGN	17%
	CODE/TEST	26%
	SYSTEM TEST	23%
	ACCEPTANCE TEST	19%
COMPUTER UTILIZATION	NUM RUNS = 108 + 150 *(KSLOC)	
PAGES OF DOCUMENTATION	DOC = 34.7 (KSLOC $^{.93}$)	

*REFERENCES: "FINDING RELATIONSHIPS BETWEEN EFFORT AND OTHER VARIABLES IN THE SEL", BASILI, FREBURGER - YAP
"PROGRAMMING MEASUREMENT AND ESTIMATION IN THE SEL", BASILI, FREBURGER

ERROR DETECTION RATE*

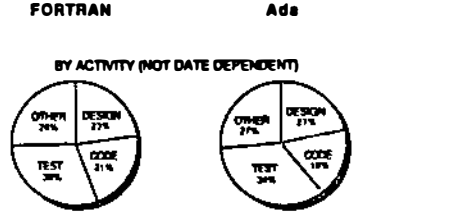
(IN ONE ENVIRONMENT)



SEL ENVIRONMENT EXPECTS TO HALVE ERROR RATE IN EACH SUBSEQUENT PHASE

*BASED ON 5 PROJECTS BETWEEN 1983 AND 1987

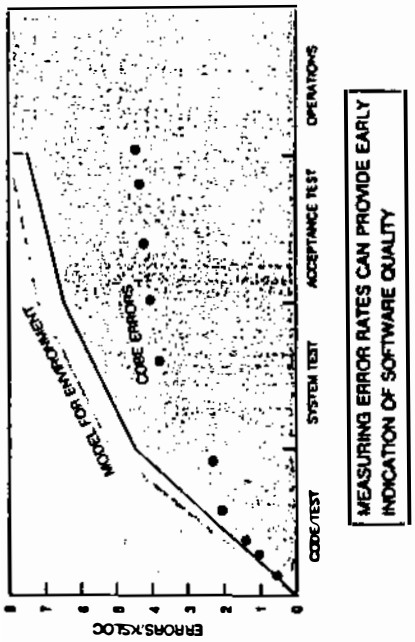
WHERE DO DEVELOPERS SPEND THEIR TIME



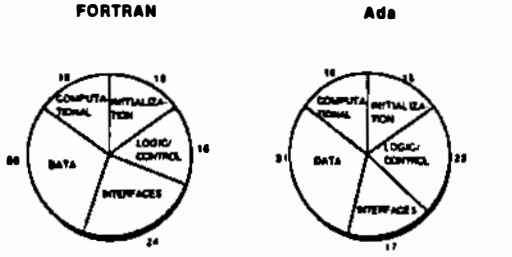
"HERITAGE" OF ENVIRONMENT IS NOT QUICKLY CHANGED

*BASED ON 5 Ada AND 8 FORTRAN PROJECTS

SOFTWARE ERROR RATES
TRACKING "COBE" RELIABILITY



CLASSES OF ERROR*



*ERROR PROFILES QUITE SIMILAR; EVEN FOR DIFFERENT LANGUAGES
*Ada SOMEWHAT FEWER INTERFACE ERRORS

*BASED ON ERROR FROM 5 Ada AND 8 FORTRAN PROJECTS

What kinds of experience can we package?

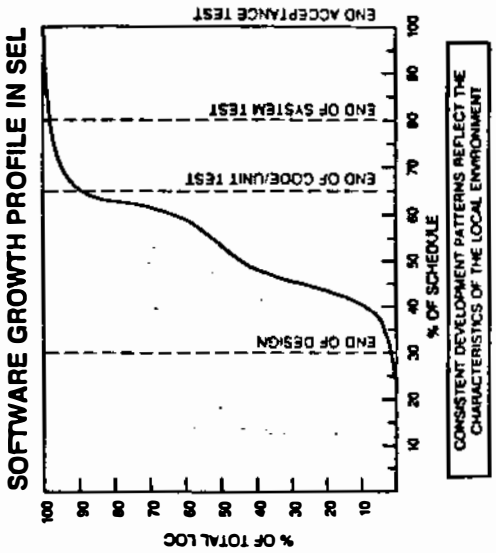
Project Characteristic Baselines and Models

Growth and change histories
- size
- staffing
- computer use
- number of test cases
- test coverage

Comparisons with the "norm"

Predictions and estimates

Guidance based upon variation from expectation



CHARACTERISTIC STAFFING PROFILES
(COMPARING 2 PROJECTS OF SIMILAR COMPLEXITY)

What kinds of experience can we package?

Products

Appropriately "parametrized" code components

Designs

Specifications

Requirements

Test Plans

What kinds of experience can we package?

Methods and Technique Definition and Evaluation

Reading vs. Testing Experiment

Functional vs. Object Oriented Design

Ada vs. FORTRAN

Process Models

SEL Standard Model for Ground Support Software
- new projects
- reuse oriented

SEL Ada Process Model

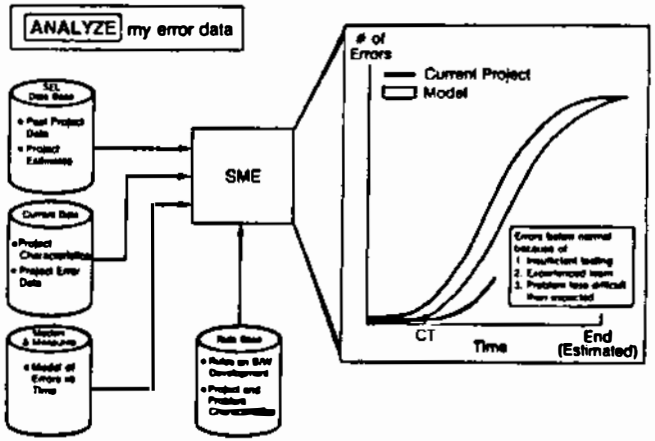
SEL Cleanroom Process Model

SOFTWARE MANAGEMENT ENVIRONMENT (SME)

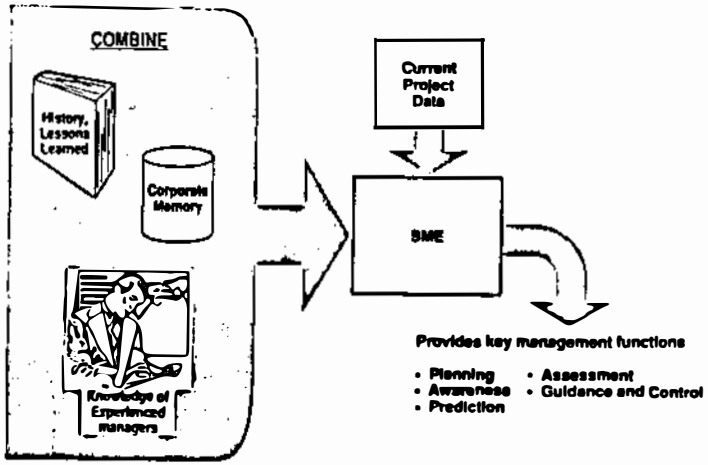
USE: (HISTORICAL PROJECTS)
- 'BASELINE' (SOFTWARE CHARACTERISTICS IN ORGANIZATION)
- ANALYSIS (MODELS, MEASURES, RELATIONSHIPS)
- 'INTUITION' (KNOWLEDGE FROM LOCAL EXPERTS)

PRODUCE TOOL(S) / ENVIRONMENT (FOR NEW PROJECTS)
- PREDICTION
- ANALYSIS
- ASSESSMENT
- GUIDANCE

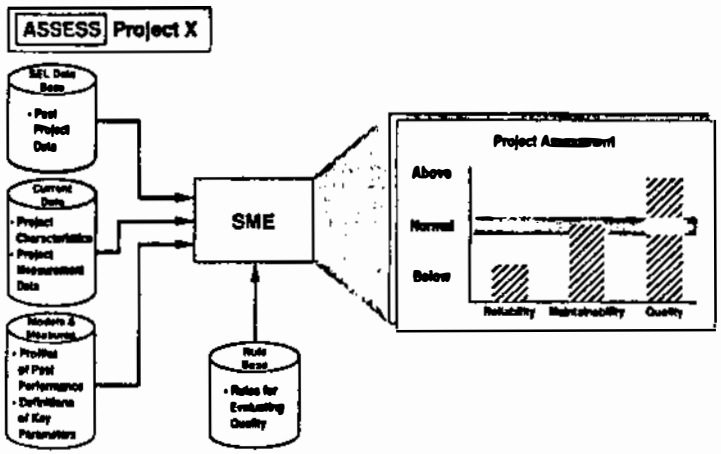
EXAMPLE OUTPUT



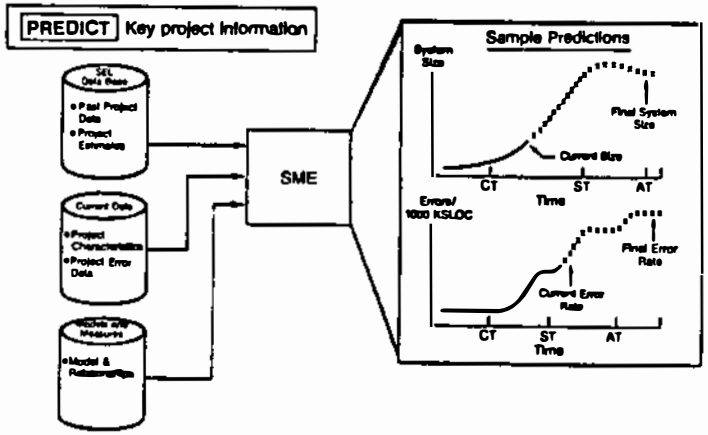
AN INTEGRATED ④ ENVIRONMENT

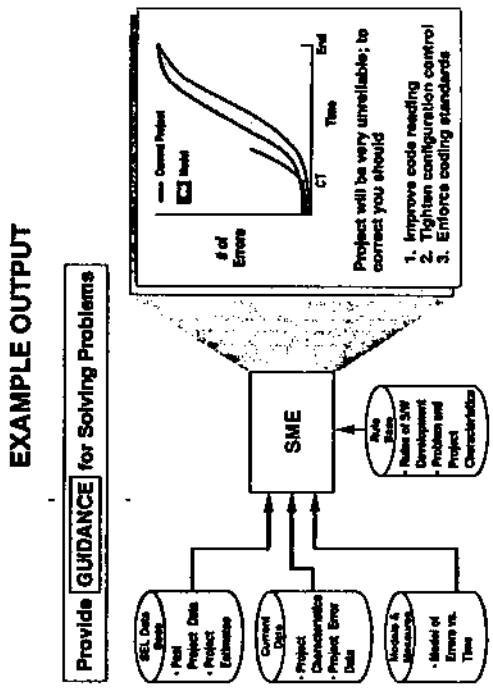


EXAMPLE OUTPUT



EXAMPLE OUTPUT





Implications and Conclusions

Experience Factory:
IMPLICATIONS

Separation of concerns/focus

Support for learning and reuse

Generates a tangible corporate asset

Formalization of management and development technologies

Can start small and expand

Links focused research with development

Experience Factory:
IMPLICATIONS

CONSOLIDATION OF ACTIVITIES
- packaging experience
- consulting
- quality assurance
- education and training
- process and tool support

FUNDING ISSUES
- separate cost centers
- corporate overhead
- project billed for packages

Costs

The level of funding depends upon the size of the program

Project data collection overhead:

Estimated to be about 5% of the project software cost (based upon SEL experience)

Must be established up-front

Typically does not effect total cost, pays for itself on first project

Experience Factory Costs:

Depends upon number of projects supported, level of effort and set of activities performed, e.g., quality assurance, process definition, education and training,...

Minimal level of effort is two people

Conclusions

Integration of the
- Improvement Paradigm
- Goal/Question/Metric Paradigm
- Experience Factory Organization

Provides a framework for software engineering development, maintenance, and research

Takes advantage of the experimental nature of software engineering

Based upon our experience, it helps us
- understand how software is built and where the problems are
- define and formalize effective models of process and product
- evaluate the process and the product in the right context
- predict and control process and product qualities
- package and reuse successful experiences
- feed back experience to current and future projects

Can be applied today and evolve with technology

Research Projects

Tame Experience Base

Resource Models
- Multi-Model Approach
- Classification Trees

Process Models
- Test Method
- Reuse Processes
  - Development
  - Maintenance

Quality Models

Component Factory
- Reference Architecture
- Care
- Qualification

Ada Experience Models
- Productivity
- Quality
- Reuse

Resource Model Research

Build a resource modeling system that integrates
- world models/top down
- local models/bottom up

and
- examines alternate modeling approaches, e.g., classification trees

World models express the general, multi-environment view of resources estimation and allocation, e.g., COCOMO

Local models express the specific development environment view of resource estimation and allocation, e.g., meta-model, classification trees

Classification trees provide a method that is
- conceptually simple to apply
- not dependent upon the underlying distribution

Provide learning capabilities so the models improve over time from
- practical application in a particular environment
- the analysis of the comparison of the multi-model environment

Toward a Test Methodology

Based upon project characteristics, project goals, the state of the project objects and prior environmental history
- generate goals for testing
- decide test phases and goal of each phase.
- decide techniques to be used in each phase

For each test phase determine
- process definition
- techniques to be used
- experience models to be used
- entry criteria
- exit criteria
- the match of entry and exit criteria for consecutive phases

Project characteristics make it possible to find similar prior projects for experience

Experience models used include
- prior defect and resource distributions associated with phases and techniques
- technique histories including strengths and weaknesses associated with various requirements,etc.

Toward a Test Methodology

These prior efforts guide the expectations of the planners on

- fault density,
- difficulty of testing various kinds of requirements,
- difficulty or cost of using various testing techniques, and
- difficulty of generating "correct" outputs for test cases.
- achieving test goals

Entry and exit criteria describe conditions that must be met before a phase can be started or ended, e.g., quantitative conditions that make assertions about the state of the object or the process, based upon definitions and goals of the phases

Information about techniques include

- tools available that support the technique
- how well the technique is understood locally
- the kinds of faults the technique is particularly good and bad at revealing

The Software Crisis isn't what we've been told it is

Robert L. Glass
Computing Trends
P.O. Box 213
State College PA 16804
USA (814)234-0728

Abstract:

Much has been written about the "software crisis" - that software is always behind schedule, over budget, and unreliable. Claims of such a crisis are often used to promote research into technology "solutions." In this talk, we look at those claims and those solutions, and find the claims largely wanting and the solutions addressing the wrong problems.

The speaker:

Robert L. Glass is president of Computing Trends, a software engineering consulting and publishing firm. He has 30 years software experience in industry and five years experience as a professor in the Seattle University software engineering graduate program. His viewpoint is that of the experienced software practitioner, not that of a manager or academician. Published work relevant to this topic includes The Universal Elixir, and Other Computing Projects Which Failed, published by Computing Trends (1977), and Software Conflict - Essays on the Art and Science of Software Engineering, published by Prentice-Hall (1991).

What is the crisis said to be?

Software is always

- behind schedule
- over budget
- unreliable

The Software Crisis Isn't!!!

But...

How come

- our financial system operates at all?
- my plane and hotel reservations worked?
- we've reached outer space?
- there's a thriving software industry?

Where'd this "crisis" come from?

Hucksters selling "solutions"
Academics seeking grants
Practitioners pushing "panaceas"

Not exactly a disinterested group!

What do they cite as evidence?

Anecdotes about spectacular failures
The GAO study data
- 60% of projects over schedule
- 50% of projects over cost
- 45% of delivered software could not be used

Does this evidence hold up under scrutiny?

YES...

there have been spectacular failures

The Software Crisis Isn't!!!

Does this evidence hold up under scrutiny?

NO...
the GAO data has been misused
- . projects studied were selected because they were in trouble!
- . of six failure causes, only was was technical (the others were in problem definition and contracting)

...and some speakers who have used this data knew these facts!

Still...

Why have there been spectacular failures?

. unstable requirements
. poor contracting

These problems are not fixable
by improving the technology!

The Software Crisis Isn't!!!

Still...

Where there's smoke, there's fire!

Even if the crisis has been oversold,
is there still a possible cause for
the prevalence of

. over schedule
. over budget
. unreliable

?

One possible cause...

Optimistic estimation

can lead to

- over schedule
  because it never was achievable
- over budget
  because IT wasn't achievable, either
- unreliable
  because we skimp on testing to try to overcome "over schedule" and "over budget"

Why do we have optimistic estimation?

Pressure by marketing and management

Technical optimism

Premature (pre-requirements!) estimation

Failure to re-estimate based on actuals

Poor estimation techniques

So where are we here...

The "software crisis" is overblown

There is a (smaller) underlying problem

The problem lies in unstable requirements, contracting, estimation

These are all customer/management problems, not software technology problems!

Solutions proposed by vendors, academics, practitioners to date CAN'T FIX IT!!!!

What do I suggest?

Quit using that expression!

Continue to work on improved technology, but for the right reasons

Pursue improvements in requirements stability, contracting techniques, estimation approaches

Because estimation is more within our grasp:
- . reference other fields for approaches
- . more history data on past estimation
- . better algorithms (!?)

What do I suggest (2) ?

Guts management!

- quit giving in to customers/management on
  - optimistic estimates
  - requirements changes without cost and schedule relief
  - difficult contract clauses
- don't COMMIT to
  - pre-requirements estimates
- update cost/schedule estimates honestly per actuals

We should get over being the "new kid on the block"

What happens then?

Short term pain

Long term credibility

The hero dies but once; the coward dies a thousand times

And remember...

Our present methods of BUILDING software,
although they can be improved,
are NOT causing any crisis.

Our present methods of MANAGING software
have allowed a crisis to be perceived.

How to Deliver 20,000 Lines of Code with only Four Defects for under $2.00 Per Line of Code

by

Norman L. Kerth,
Elite Systems

Robert Rhodes &
Wacker Siltronic

John Burley
Victory Software Systems

Abstract

Our experience shows that continuous process improvement, a well known concept in the manufacturing sciences field, is applicable to the process of developing software. We show how this has been applied to the specification phase through the use, refinement, modification and invention of models. The motivation for this evolution of modeling is thorough the analysis of errors and rework in our product. Errors and rework not only demonstrate a deficiency with our product but also with the process that we used to create the product. We use this analysis to understand the deficiencies with our models and how they should be changed.

Biographical Sketches

Norm Kerth has been consulting full time in the Portland area since 1984. He specializes in disciplined software engineering and training activities. His projects have included the entire life cycle (specification, design, implementation, and testing) and responsibilities have included either or both engineering or project management. His clients include Mentor Graphics, Tektronix, Data General, Sequent, Advanced Data Products, Intel, Systems Technologies Corporation, Data Dynamics, Leupold & Stevens, Wacker Siltronic, Advance Computer Engineering, Josten's Learning, the University of Southwestern Louisiana and the National Science Foundation. He teaches for Portland State University at OCATE.

Before starting his own consulting company in 1988, Norm held the position of Principal Engineer/Research Assistant Professor in the University of Portland's Applied Research Center. He is an expert in the areas of specification and design methodologies and CASE tools. He has special interests in the areas of specification for human interface intensive problems and bringing engineering discipline to the practice of developing object-oriented programs. Before joining the U. of P., Norm spent ten years working for Tektronix in a variety of positions.

Elite Systems
7400 S. W. Barnes Rd. #911
Portland, OR 97229
(503) 297-8677

Robert Rhodes has been with the Wacker Corporation for eight years as a computer engineer and a control systems engineer, working in the fields of real-time control of industrial processes and measurements, automated process specifications and monitoring, systems analysis and design, distributed data base design, and project management.

John Burley is the president of Victory Software Systems, which has provided software development services to major companies in the Portland area since 1981. John has been developing software for almost 19 years since his high school days. His interest in structured methodologies started in the late 70's when he attended his first class on Tom DeMarco's Structured Analysis. He is an active member in the Pacific Northwest Software Quality Conference's Program and Workshop committees. He is currently studying, designing and implementing object-oriented software solutions.

How to Deliver 20,000 Lines of Code with only Four Defects for under $2.00 Per Line of Code

by

Norman L. Kerth,
Elite Systems

Robert Rhodes &
Wacker Siltronic

John Burley
Victory Software Systems

Introduction

Peter Freeman declares " The 'software crisis' is dead. 'When did it die?' you might ask. Many may have not noticed ... nonetheless, it is clear that things are different today than they were a few years ago in the realm of software development." [Humphrey] Aggressively, he argues that we can build reliable software on time and within cost and then lets Watts Humphrey explain how. If you read one professional book this year, let it be "Managing the Software Process."

Watts Humphrey's book is based on a concept that is well understood in manufacturing environments - the idea that any process used to produce something can be and should be continually improved. He observes that there is a process to developing software (even if you are not aware of it), and proceeds to develop a plan for systematically understanding your process and improving the weakest parts of it.

For some time now, we have been studying the process of developing software. We have asked what works well and what should we do differently next time. Over time we have seen our results improve. Recently, we developed a program for Wacker Siltronic, in an environment where the process could be well monitored. The program, used for scheduling and definition of control for a manufacturing process, was implemented in Object Pascal and the compiler reports the size to be over **20,000 lines**. During the coding phase, the programmers found **5 defects in the specification and design**. During testing, only **4 programming defects** were found. To date, the **end-users have found only 4 defects**, all of which were discovered during the final acceptance test. **No defects have been found since the program went on-line.** Work on the specification, high level design and test suite generation involved 175 hours. Low level design, coding, debugging, testing, rework, archiving, documenting, etc. consumed 600 hours. But these metrics do not communicate how much confidence we had in knowing what the program should do, knowing that there were no loose ends, and knowing that we had identified and resolved the high risk issues early in the project. The metrics do not show the control that we had over the project and how project difficulties became apparent quickly and were resolved by more disciplined use of our process.

A strong motivation for the development of our process is to minimize rework. We have accepted without proof that rework is our key measurement for determining the effectiveness of our process. We manage the project in such a manner that every action we take supports the notion of minimizing rework. Our focus on rework is not new, it has been a mainstay of manufacturing science for quite some time. We include as rework: defect repair of the code, debugging, accommodating design change, refining and revising specifications, etc. Of course, reality requires some of these activities, but we strive to keep them to a minimum. Furthermore, rework does not give us an excuse for paralysis, where we minimize rework by making no decisions or progress. Our process has to make progress and progress has to be visible.

When we talk about "our process," the key word is *our*. The process that works well for one team on one class of projects will not be effective in another environment. Just as each manufacturing process needs to be tailored to the specific environment [Goldratt] (market, product mix, people and machine resources, etc.), each software team needs to accept the responsibility for defining, analyzing and improving the process that they use to create software. All too often, the process is ignored in favor of the product. All too often, software engineers let or expect their managers to take care of the process while they focus on their own piece of the project. All too often, things don't quite come together at the end of the project and we slip the schedule and/or ship a product of which we are not proud. And all too often, we do not learn from our mistakes and improve how we work in the future. Repeating the message of this paragraph - **each team member needs to accept the responsibility for defining, analyzing and improving their own software development process.**

In the rest of this paper we talk about one part of our process - specification development. The necessity of developing fine specifications is most important in reducing rework. Specifications can set a clear direction for the design and implementation. They can minimize the effort in developing test suites and developing manuals. They can identify high risk issues that need to be managed closely and scheduled early. [Boehm] has shown that defects in the specification are most costly to repair, so extra effort in getting the specification complete, consistent, correct and coherent is a good investment.

Software Specification as Part of the Process

Anyone who has taken a software engineering course has heard that one needs to develop a specification as an early part of the software development process. They have seen the waterfall model diagram and heard that specifications describe "what" the system should do but not describe "how" to do it. If not from such a course, then one is likely to hear a request for a specification from their manager.

The problem is that we have not been taught how to think about and approach problems in such a way that we can write good specifications. Most specification documents are written at a time when we have ideas to be put into a new product but have spent very little effort understanding the interrelationships between these ideas and concepts. Typical specification documents might contain design detail at some points in the product and other aspects will not be mentioned. In the better run projects, a walkthrough will be held. At this point, the specification is likely to be 80 pages of the most boring English text (the people chosen to write the specification where selected because they were good at something other than writing). After a great deal of effort and concentration, several holes will have been found and some of the inconsistencies will be identified but there will still be an uneasy feeling that not all has been understood. But we shrug and figure that we will figure it out as the project goes along; after all, we did last time.

With some projects, there are engineers who can't wait for the specification to be completed so they start coding a "prototype." Soon managers and marketeers see the results of this prototype and get excited. Assumptions are made that the projects is further along than it is really, and pressure develops to "evolve the prototype into a product."

Proceeding from either of these points puts the project in a vulnerable state, since there is momentum towards a solution when the understanding of the goals are not clear. In such a situation, decisions will be made on-the-fly, and will rarely take into account all

the situations or states that need to be considered. Some of these decisions will involve rework which is costly financially, costly emotionally and error-prone. Since the understanding of the project is not complete, there is the possibility that high-risk issues will go unobserved until late in the development process. In short, the project is not in management's control, and schedule estimates are likely to be erroneous.

Attempting to bring more structure to the specification phase, Tom DeMarco explained to us how to use data flow diagrams(DFD's). Unfortunately, we know only a few people who have actually used DFD's on more than one project. We believe people tried DeMarco's techniques but did not find them to be the perfect answer and discarded them. This is unfortunate, since DFD's are a good foundation and have been studied, extended, augmented, and supplemented by many researchers for over a decade [McMenamin] [Shlaer] [Ward] [Hatley] [Weinberg] [Gause] [Kerth]. This collective work provides us with a great deal of wisdom about developing complete, consistent, coherent and correct specifications.

A common characteristic in all this work is the use of models to represent some aspect of the system, while intentionally deferring concern for other aspects until later. Each of these models describes some aspect of the system, and focuses one's thinking about the problem at hand. These models usually have some graphic representation as well as supporting documents. Well defined models provide a way to discover incompleteness and inconsistency.

After years of experimentation, we have found no one methodology or system modeling technique to be effective on real life problems. And yet, we have experienced value in a number of methodologies and have found that combining methodologies, changing methodologies and inventing our own, does produce a process that is very effective. Our process includes four major modeling techniques:

Methodology Name	Purpose	Remarks
Information Modeling	Used to identify information dealt with by the users within their problem domain. It is a way for programmers to structure the user's vocabulary without being obnoxious.	This method is often used in data base work, to structure information in a data base. We have used these same concepts but deploy them towards the data relationships found in the end-users' environment and vocabulary.
Finite State Machines (FSM's)	Used in two distinct ways: 1) to describe the behavior of the system and how that behavior changes due to specific conditions, 2) to define how the nature of some knowledge changes due to specific conditions.	FSM's have been used to explain system behavior for quite some time but this second use, to understand how information changes, grows and becomes obsolete or not needed, is not often exploited.
Event-Partitioned Data Flow	Used to understand all the data that flows and the impact to data stores when an event occurs in the system. It addresses what the needs of the user are while leaving for later the difficult issues of how to meet the user's needs.	This methodology is a distant cousin to Structured Analysis. The notation is the same (thus most CASE tools can be used) but there is a shift in one's thinking about analyzing systems. Our thinking is response driven, stimulated by an event issued by either a user or a piece of hardware (e.g., an interrupt).
Three Dimensional Human Interface Perspective (3D-HIP)	Used to express the human interface and consider the appropriate response for every stimulus possible from the user for every possible state.	This is an original methodology composed of: views (i.e., graphical representations), a view flow and operational specifications (op-specs).

These modeling techniques systematically add information to our understanding of the program to be developed. Progress in later models is based on decisions noted in earlier models. Consistency checks between models are established and maintained as changes occur.

At this point in the paper, we will discuss each of these modeling techniques at an introductory level. To illustrate these models we will present portions of a bug tracking system. This system is designed to capture data about defects found in a product and track the resolution of those defects. In some cases a defect will need to be deferred, or

will be determined to not be a defect. In other cases the defect will be scheduled for repair. Given these circumstances, the defect will be traced through the repair and through a quality assurance re-testing activity. Status of defects in a summary report format is also important to the users of this system.

Information Modeling

The first modeling technique, information modeling was discussed by [Shlaer], but we deploy that work differently. Often information modeling is used to express information that is to be incorporated into a data base. However we have used this modeling technique to understand the information that exists in the problem domain. It is a model that shows how pieces of information are related and how changes to some pieces effect other pieces. Many engineers confuse the capture of this information with developing a solution to some expressed need. At this point in the specification process, we are simply understanding what the problem is.

Early in the life of a project, a great deal can be learned by thinking about the obvious. For example, if you are working in a problem domain where the user wants to see a waveform, one might get a much better understanding of the problem by asking "what information makes up a waveform." I'm not talking about the green line on a scope screen but about the information associated with a waveform from the user's perspective (more generally stated - the information associated from the producer's and consumer's perspectives). From this introspection we may discover that a waveform not only has time-ordered points of magnitude, but might have peaks, valleys, inflection points, slope, attenuators in several dimensions, carry special identification (e.g.,. implemented with coloring or cursors) at certain points, and may not be time ordered at all! Until one understands the information relationships in the problem domain, they do not understand the problem and can not develop, with confidence, a solution to any aspect of the problem. We have heard managers ask us to "build the right product and build the product right". An information model is needed to understand what the right product is.

When we begin to think about the problem of tracking bugs, and ask "what information is associated with a defect?" We might talk with a few people, look at a few defect reporting forms, and decide that information associated with a particular defect report might include an identification number, a date the defect was discovered, a contact person, etc. We also might come to understand that certain summaries of collections of bugs or "reports" are needed to manage a software process at various times. The information model shows this information:

INFORMATION MODEL OF A BUG TRACKING SYSTEM

bug =		id + date + submitter + priority + summary + description + personResponsible + status + descriptionOfRepair
status =		[submitted | scheduled | fixed | stillFlawed | aClosureState]
aClosureState =		[verifiedFixed | notABug | deferred]
report =		{ id + ({optionalField}) }
reportDescription =		{searchCriterion } + {sortCriterion} + ({optionalFieldId})
optionalFieldId =		[date | submitter | priority | summary | description | personResponsible | status | descriptionOfRepair]

searchCriterion = [idCriteria | dateCriteria | submitterCriteria | priorityCriteria | personResponsibleCriteria | statusCriteria]

sortCriterion = [idCriteria | dateCriteria | submitterCriteria | priorityCriteria | personResponsibleCriteria | statusCriteria]

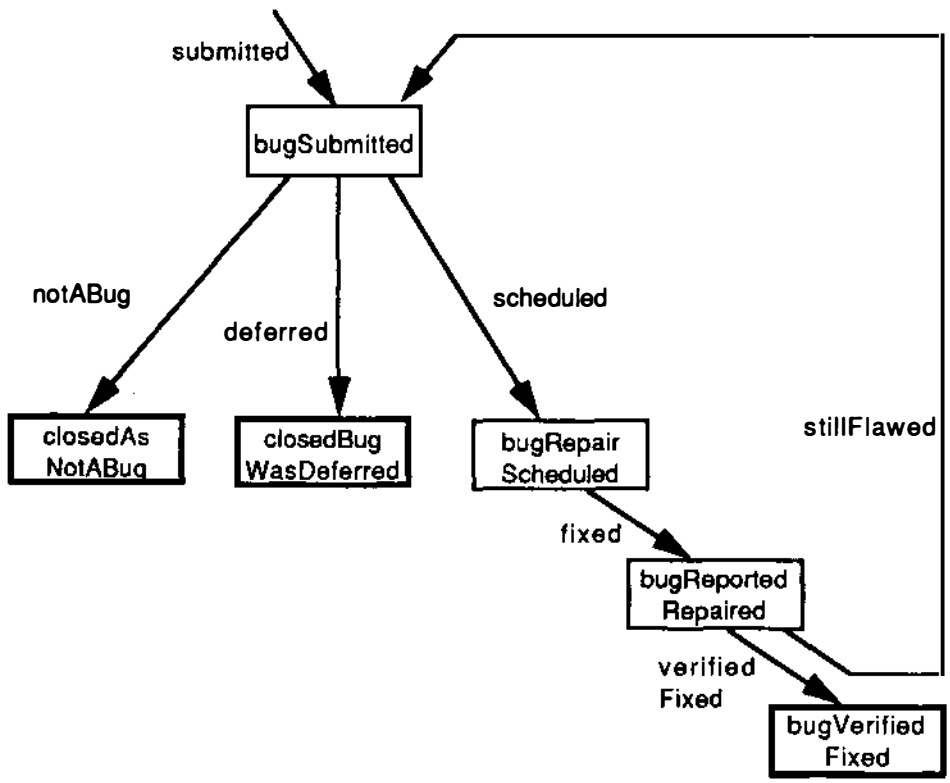
In this notation, the = sign is pronounced as "is composed of", and the + sign is pronounced "and". So we have "a bug is composed of an id and a date and a submitter..." The { }'s are pronounced "some number of" , the [|]'s means "one of the following and ( )'s imply optional information. So we can see that a report is composed of some number of id's and for each id, there may be some number of optionalFields.

While developing an understanding of the information, a piece of information will sometimes be found where its content and meaning will change over time. For these pieces of information, the nature of change is as important to understand as is the circumstances under which the change occurs. For this, we use a finite state machine.

Finite State Machines Applied to Information

Our bug tracking problem provides us with a good example of a kind of information that needs to be modeled with a FSM. When a bug is first reported, the id, date, submitter, summary and description need to be included, but it is not likely that the personResponsible data would be available until management has decided if the defect is to be repaired. Likewise, the descriptionOfRepair data would not be available until the defect has been removed.

We can note these characteristics of a bug with the following FSM model:



FINITE STATE IMPLICATIONS OF A BUG

Condition or State	Definition of Information
status = submitted	id + date + submitter + summary + description
status = scheduled	id + date + submitter + priority + summary + description + personResponsible
status = fixed	id + date + submitter + priority + summary + description + personResponsible + descriptionOfRepair
status = stillFlawed	id + date + submitter + priority + summary + description + personResponsible + descriptionOfRepair /* Note: the tester may add information to the description of the flaw */
status = verifiedFixed	id + date + submitter + priority + summary + description + personResponsible + descriptionOfRepair
status = notABug	id + date + submitter + summary + description /* Note: that the rational of why this is not a defect may be added to the description */
status = deferred	id + date + submitter + priority + summary + description /* Note: the description may have information added to it to explain why the defect was deferred and how long. */

Now the designer of a real bug tracking system may object to some of the decisions that we have made about how information is captured and stored. That makes sense; this is a toy problem. But what is important is that we have developed a model that can be read, understood and discussed very early in the process of developing a bug tracking system. Avoiding this model, but making the same decisions increases the amount of rework much later in the process.

Event-Partitioned Data Flow

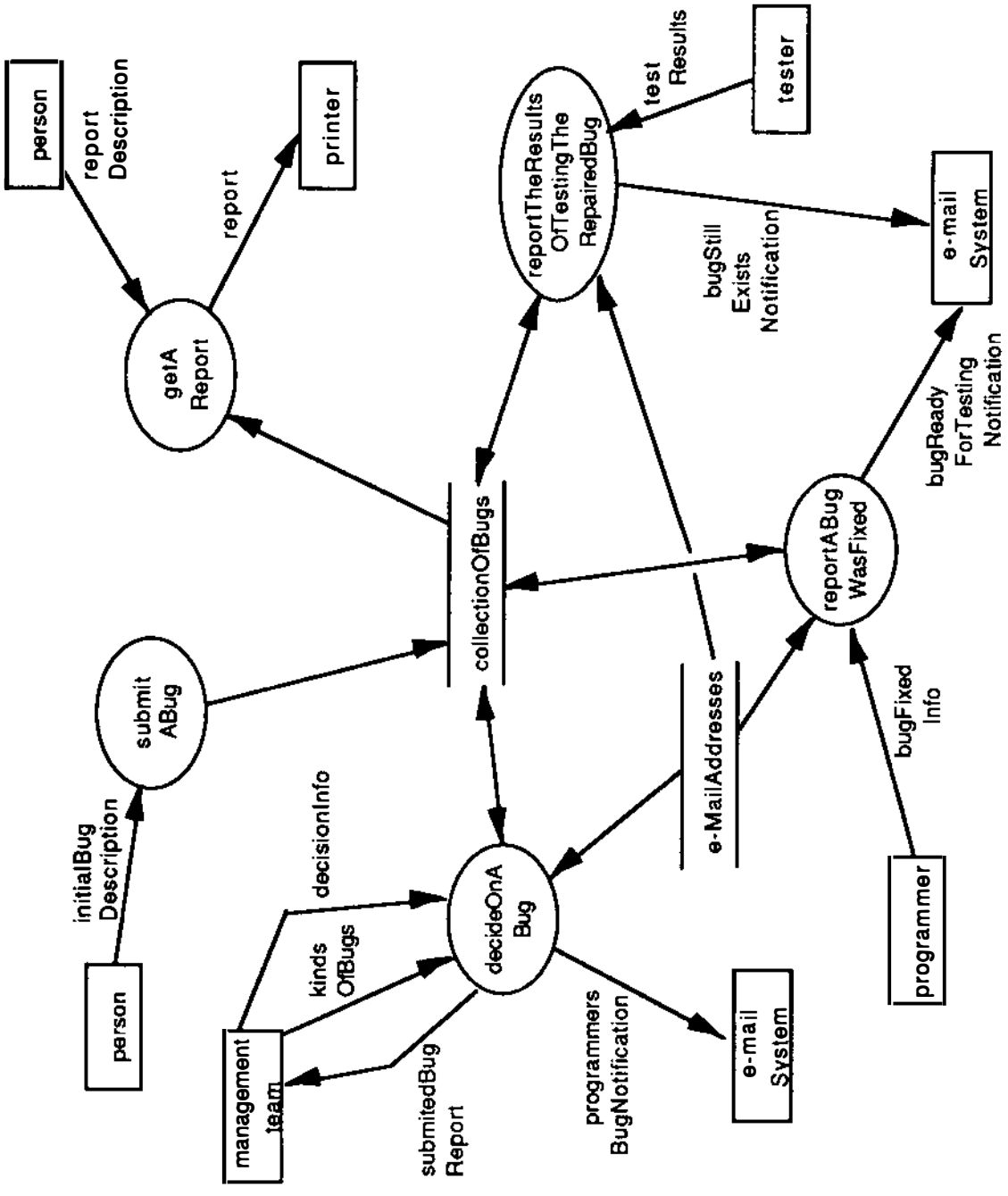
Once the information model and the necessary finite state machines have been developed, our attention can focus on the solution domain. For this we use a variation on structured analysis discussed by [McMenamin] called *event-partitioned data flow.* In this model, we identify an activity or goal that a user turns to our system to accomplish. For each of these activities, we identify the information necessary from the user and the information provided to the user. These activities are called an *events* (not to be confused with the Smalltalk or Macintosh events such as mouse clicks or keyboard entries).

Examples of events, from our bug tracking example include:

submitting a bug report,
deciding if a bug is to be repaired, deferred or is just not a flaw,
reporting that a bug has been repaired,
reporting on the results of testing a repaired bug, and
getting a summary report about bugs in our system.

We now develop an event-partitioned data flow diagram with supporting data dictionary and mini-specifications (see [DeMarco] for details if structured analysis is not known to you). For every event that we have found, we create a process bubble. There are no process bubbles that are not events. The result is a data flow diagram that has processes interacting with users or other external entities and data stores. Note that data never flows from bubble to bubble! Each event performs all the work necessary to accomplish the work that the user set out to accomplish, thus there is no reason for data to flow to another event.

Returning to our bug tracking system, here is the event-partitioned data flow:



Along with the event-partitioned data flow, we create a logical data dictionary, mini-specifications and a physical data dictionary. A logical data dictionary describes the actual data structures that flow between entities or that are stored in a data store. Much of this definition is fed by the work that we did on the information model, but there is not a one-for-one correlation. Instead, we make decisions in the solution domain based on our deep understanding of the problem domain.

LOGICAL DATA DICTIONARY

bug =	id + date + submitter + priority + summary + description + personResponsible + status + descriptionOfRepair + (discussionOfTestResults)
collectionOfBugs =	{ bug }
initialBugDescription =	summary + description
submittedBugReport =	id + date + submitter + summary + description
decisionInfo =	["notABug" | "deferred" | "scheduled" + priority + personResponsible]
programmersBugNotification =	e-MailHeader + id + date + submitter + summary + description + priority
bugFixedInfo =	id + descriptionOfRepair
bugReadyForTestingNotification =	e-MailHeader + id + date + submitter + priority + summary + description + personResponsible + status + descriptionOfRepair
testResults =	id + ["verifiedFixed" | "stillFlawed" + discussionOfTestResults]
bugStillExistsNotification =	e-MailHeader + id + date + submitter + priority + summary + description + personResponsible + status + descriptionOfRepair + discussionOfTestResults
reportDescription =	{searchCriterion } + {sortCriterion} + ({optionalFieldId}
report =	id + ({optionalFie
optionalFieldId =	["date" | "submitter" | "priority" | "summary" | "description" | "personResponsible" | "status" | "descriptionOfRepair"]
searchCriterion =	[idCriteria | dateCriteria | submitterCriteria | priorityCriteria | personResponsibleCriteria | statusCriteria]
sortCriterion =	[idCriteria | dateCriteria | submitterCriteria | priorityCriteria | personResponsibleCriteria | statusCriteria]
e-MailAddresses =	{ person + e-MailHeader}
status =	["submitted" | "scheduled" | "fixed" | "stillFlawed" | aClosureState]
aClosureState =	["verifiedFixed" | "notABug" | "deferred"]
personResponsible =	person

For each event, we develop a mini-specification. A mini-specification describes how all the data acquired from the user is used, describes what other data is acquired from other entities or data stores and how that data is transformed, stored and/or presented to the user or other entities.

MINI-SPECIFICATIONS

```
submitABug
    Collect the initialBugDescription information from the user;
    Get the date and the user's name from the system;
    Use this information to create a new entry in the collectionOfBugs file.

getAReport
    Get the reportDescription from the user;
    Search collectionOfBugs for each bug that meets the reportDescription's
    searchCriterion (i.e., criteria);
    Order these bugs according to the reportDescription's sortCriterion (i.e., criteria);
    Produce the report by dealing with each bug in order -
        print the fields that are described in reportDescription's optionalFieldId's

decideOnABug
    Search the bugs in the collectionOfBugs file, finding all the bug with the same
    status as included in kindsOfBugs
    FOR Each of these bugs
        Display them to the management team;
        Get the decisionInfo from the management team;
        IF the decision is "notABug" or "deferred" THEN
            update the bug's status field of the bug with "notABug" or "deferred"
                respectively;
            Return this updated bug back to the collectionOfBugs file;
        ELSE the decision is "scheduled" SO
            Set the bug's status field to "scheduled";
            Set the bug's priority and personResponsible fields to the information
                supplied by the management team;
            Use the personResponsible to look up the e-MailHeader in the e-
                MailAddress file;
            Construct a programmersBugNotification and pass it to the e-MailSystem;
            Return this scheduled bug back to the collectionOfBugs file;

reportABugWasFixed
    Get the bugFixedInfo from the programmer;
    Using the id, get this bug from the collectionOfBugs file;
    Update the status field;
    Use the data from this bug to construct a bugReadyForNotification;
    Use the id "Tester" to look up the e-MailHeader in the e-MailAddress file;
    Construct a bugReadyForTestingNotification and pass it to the e-MailSystem;
    Return this fixed bug to back to the collectionOfBugs file;

reportTheResultsOfTestingTheRepairedBug
    Get the testResults from the user;
    IF "verifiedFixed" THEN  using the id, change the status of the bug in the
    collectionOfBugs to "verifiedFixed"
    ELSE "stillFlawed" SO
        Using the id, change the status  to "stillFlawed"
        and add the discussionOfTestResults to the bug in collectionOfBugs;
```

The physical data dictionary, defines the exact type of each of the lowest level pieces of data.

PHYSICAL DATA DICTIONARY

id =	Integer
date =	Day-Month-Year date format
submitter =	String < 32 characters
priority =	Integer, range [1-5]
summary =	String < 64 characters
description =	String < 4096 characters
descriptionOfRepair =	String < 4096 characters
discussionOfTestResults =	String < 4096 characters
person =	String < 32 characters
e-MailHeader =	String < 128 characters

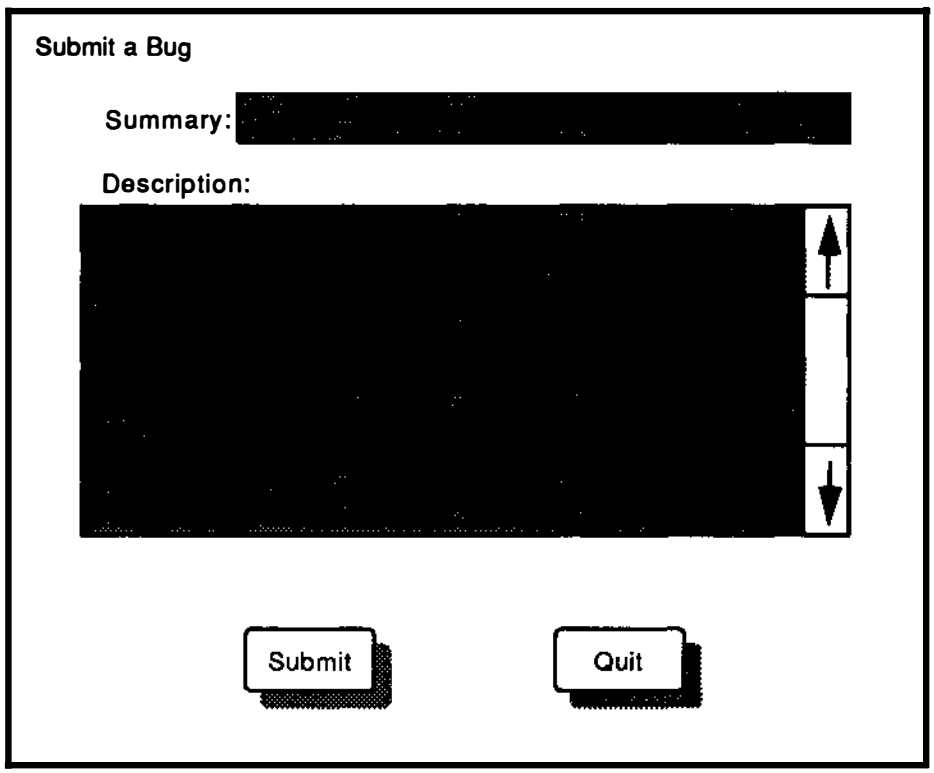
Upon completing the event-partitioned data flow, we have said a great deal about information coming from a user, going to a user and how the data gets processed, but we are not yet ready to design a solution.

Sun Microsystem's Bill Joy has argued that as much as 75% of today's application systems are devoted to human interface. If one proceeds to the design phase without specifying the human interface, then a great deal has been left unsaid about WHAT the program will do. Again, this will contribute to significant rework costs and effort in the later stages of the project (less rework but an unsatisfactory human interface, is another option).

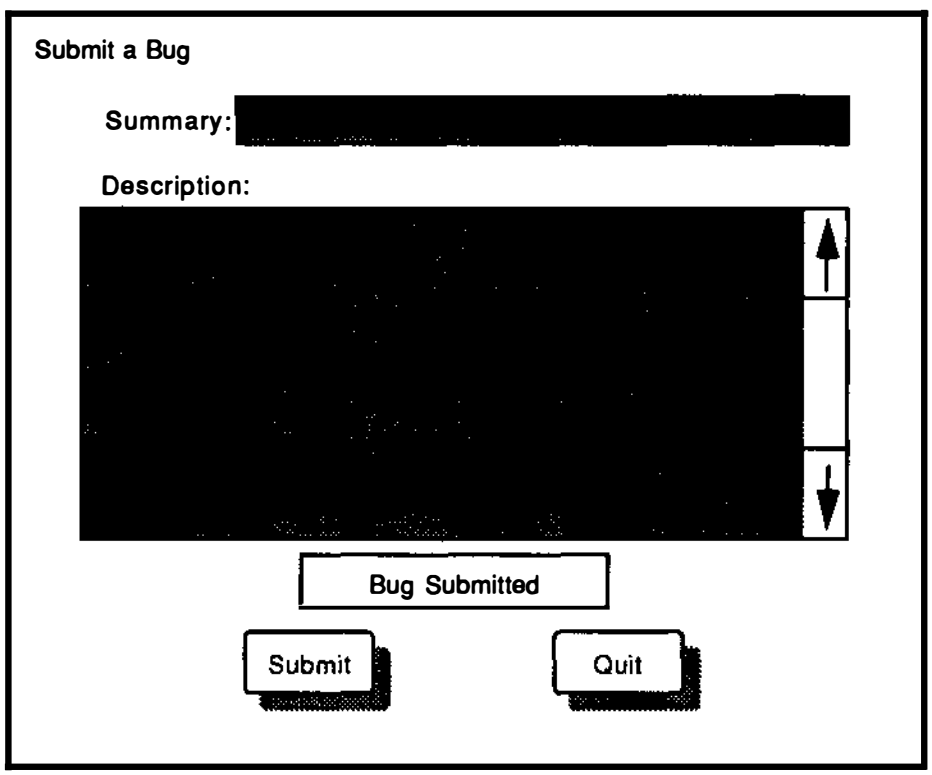
Three Dimensional Human Interface Perspective

To resolve the human interface issues, we use a **Three Dimensional** Human Interface Perspective (3D-HIP)[Kerth]. The three dimensions include: 1) a collection of graphical representations or "views" seen by and manipulated by the user, 2) a "view flow" or finite state machine that shows which views follow from a particular view upon receipt of a specific stimulus, and 3) a collection of operational specifications that document the appropriate response for all stimulus in each view.

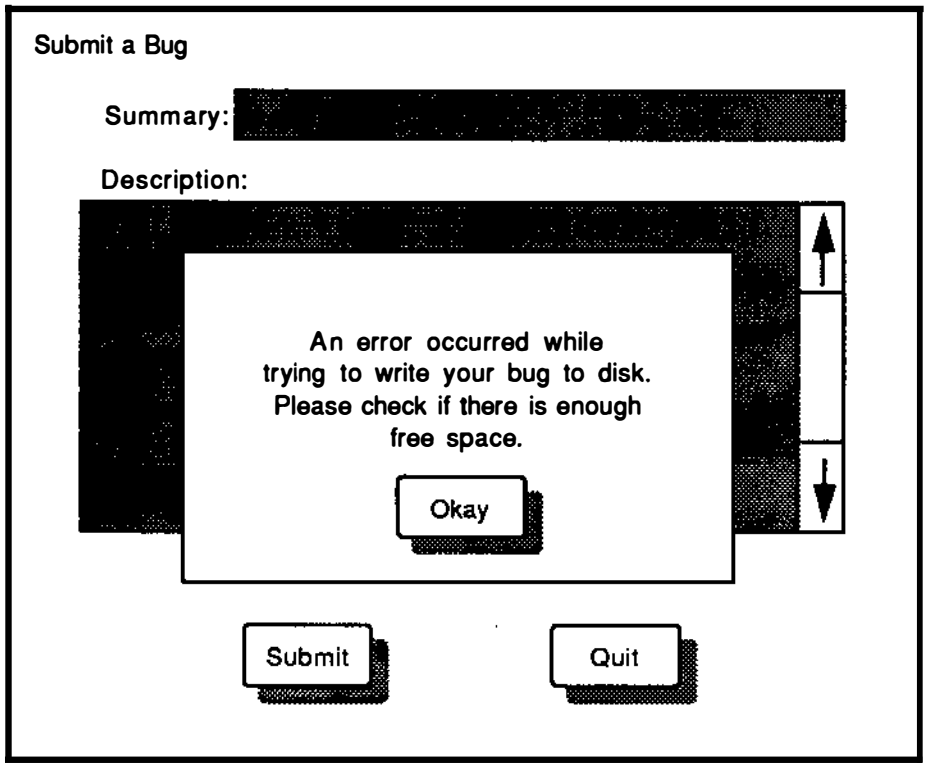
Given the amount of space required to represent a human interface, we will present only part of the 3D-HIP specification; the part for submitting a bug. First we see four views:



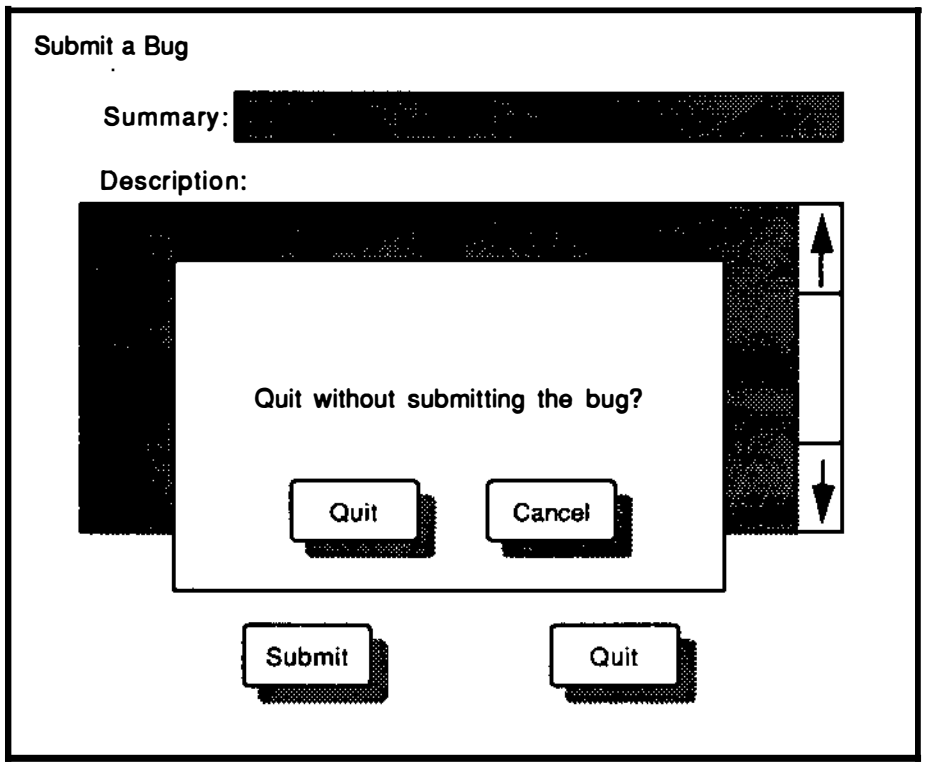
View 1.0 - Submit a bug



View 1.1 - Submit a bug



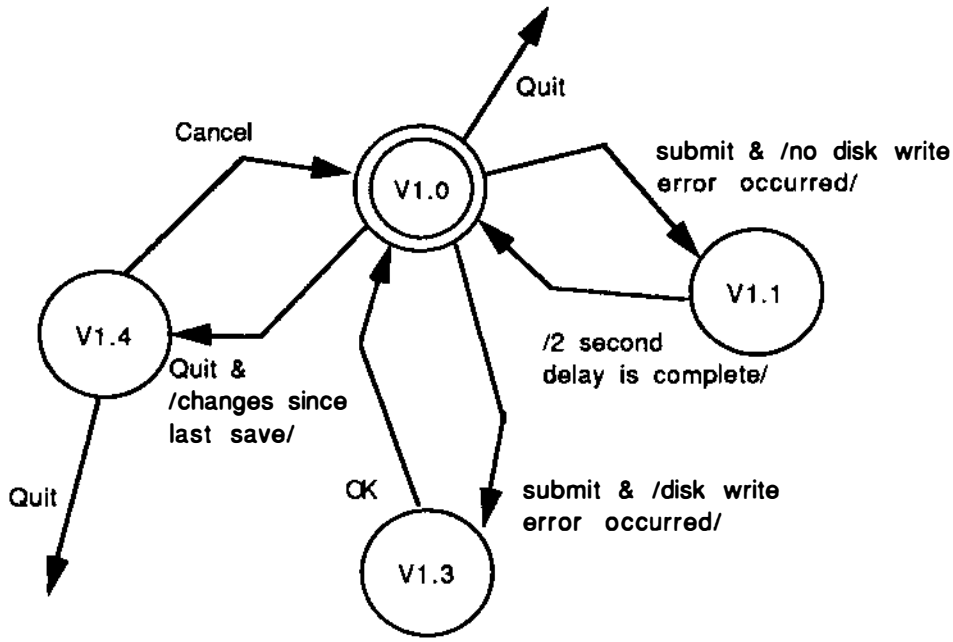
View 1.3 - Submit a bug



View 1.4 - Submit a bug

Each view represents a variation of the basic view 1.0. These views try to show pictorially what the interface looks like. Often these views are representative, rather than exact. One purpose of these views are to give a potential user an idea of what is intended to be built. At this point feedback begins. The reader will also note that we have identified every error message one might expect from the application. This forces the designer to think through the unattractive behaviors as well as the normal behaviors. By seeing these views long before coding starts, one can assure that dialog messages and application control is common across the program.

Beyond the views, we need to document under what circumstances the program will move from one view to another. This is explained in the viewflow:



Viewflow

In the view flow, each arc shows the conditions that caused one view to be replaced with another. If the condition is a stimulus from the user then the notation is just the name of the stimulus. However, there are times when the view flow is influenced by some other condition. We note this by including the description within / /'s, for example the bug tracking system moves from V1.1 to V1.0 once a */2 second delay/* stimulus has been achieved. The double circle is the starting view.

In addition to the views and the viewflow, we need to describe the appropriate response for every possible stimulus for every possible state that might influence such behavior. This is done in an operational specification, or op-spec. Even though it is quite lengthy, the op-spec is such an important document that it included at this point in the paper:

Operational Specifications
(Op Specs)

View 1 - Submit a Bug

Entry Operations:

Create a view titled "Submit a Bug";
Create a labeledFillout titled "Summary:" with a text field of 64 characters,
Set its origin to 5,5;
This object is called summaryField.

Create a scrollableEditBox titled "Description:"with a text field of 4096 characters,
Set its origin to 5,9; Set its extent to 69,19;
This object is called descriptionField.

Create an actionButton Titled "Submit";
Set its origin to 30, 23;
This object is called the submitButton.

Create an actionButton Titled "Quit";
Set its origin to 50, 23;
This object is called the quitButton.

Set the summaryField labeledFillout to active;
Display the view.

Commands passed to the Human Interface Components

Stimulus	State	Response
any key	active field recognizes it	Active field executes it
	active field doesn't recognize it and view does not recognize	Beep.

View Recognized Stimulus

Stimulus	State	Response
<up arrow>	summaryField is active	Move cursor to the last line showing of the descriptionField; Activate the descriptionField; Deactivate the summaryField

	descriptionField is active and <up-arrow> is not presently valid	Deactivate the descriptionField; Activate the summaryField
	descriptionField is active and <up-arrow> is presently valid. (Note that this stimulus & state are covered by the *Commands passed to the Human Interface Components* section above)	Perform scrollableEditBox behavior
<down arrow>	summaryField is active	Move cursor to the first line showing of the descriptionField; Deactivate the summaryField Activate the descriptionField;
	descriptionField is active and <up-arrow> is not presently valid	Deactivate the descriptionField; Activate the summaryField
	descriptionField is active and <up-arrow> is presently valid. (Note that this stimulus & state are covered by the *Commands passed to the Human Interface Components* section above)	Perform scrollableEditBox behavior
submitButton clicked	all states	Perform submitting operations; IF an error occurred THEN Display the message on view 1.3; On "Okay", remove the message from the screen; ELSE no error SO Display the message on view 1.1 for 2 seconds and then remove it; Clear the text from the summaryField; Clear the text from the descriptionField;
quitButton clicked	summaryField AND descriptionField are clear	Perform exiting operations.

	summaryField OR descriptionField are not clear	Display the message on view 1.4; IF quit THEN Perform exiting operations ELSE cancel SO Remove message from the screen; Ignore the stimulus;

Submitting Operations:

Create a new bug;
Move the text from the summaryField into the bug as its summary;
Move the text from the descriptionField into the bug as its description;
Get the date and the user's name from the system and enter them into the bug as date and submitter, respectively;
Save the bug in the collectionOfBugs.

Exiting Operations:

Clear the screen;
Deallocate the view titled "Submit a Bug";
Deallocate summaryField;
Deallocate descriptionField;
Deallocate submitButton;
Deallocate quitButton.

Our approach for developing 3D-HIP documents starts by studying the information flow between the user and the each event, diving into the mini-specs and data dictionary as needed to discover the required inputs from the user. Then we construct a set of views that acquire and present the required and desired information. The viewflow follows and when work begins on the op-specs, many views dealing with error conditions or warnings to the user are added to the collection of views.

Upon completion of the four models, the software engineer usually has a good idea of what the application must do. **Developing these models seems like a great deal of work. Frankly, it is, but it is not extra work!** There is no decision made in any of these models that would not have to be made by a programmer during coding, if not made earlier. But at that point in time, the programmer is dealing with a great many issues related to how to express algorithms and data structures effectively in a programming language. His/her attention is divided and the potential for introducing defects, or simply forgetting about some aspect of the system is increased. Furthermore, while coding, the programmer is dealing with details and can not take the time to develop a broad perspective to support consistency across an application, leading ultimately to rework.

These four models have been used on several real-life projects, ranging from 10K lines of code up to 250K lines, and the results have been quite encouraging. In the next section, we discuss our observations about applying them to Wacker's Section 5 Scheduling Manager.

Experiences Using the Process

As we mentioned above, we have experienced very few defects. The Section 5 Scheduling Manager is about 20,000 lines. During coding, the programmers found 5 defects in the specification and design. During testing, only 4 programming defects were found. To date, the end-users have found only 4 defects.

But we have several other interesting observations:

As with many projects, this one fell behind schedule (we initially underestimated the effort to develop the human interface objects while a programmer is working though the object-oriented learning curve). In response to our schedule difficulties, we focused more on our process, rather than discard it. We used more walkthroughs, we drew more precise design diagrams, and our communications and management decisions were driven by the 3D-HIP and design documents. As a result, the team communicated better, worked better, our schedule estimates became more accurate. Furthermore, the project leader increased his understanding of the status of the project and the difficulties we were experiencing.

Adhering to the notion of continuous process improvement, each time a defect was found, the project slipped, or some other symptom that our software development process was not perfect, we took the time to understand what was wrong with our process and how to improve it. As the project progressed the schedule slips ceased. Given a chance to develop a similar sized system, we believe this team would not create the same classes of errors that were found in the specification and code.

To meet our deadlines, we added a programmer to the project in the middle of implementation and felt almost no impact to the team's productivity as he worked through the learning curve. The 3D-HIP and design documents, combined with our existing code and a two hour team discussion was all that was required for training. In about a week, we had doubled our programming capacity.

We also found that the programmer productivity rate continued to increase throughout the implementation, which was directly proportional to the programmer's ability to reuse previous design and code segments. As time went on, we had more that we could reuse. Design segments developed in this project have since been ported to another machine and used in other projects.

The op specs provided excellent input for the development of a comprehensive test suite. So much so, that developing written test suites was rather easy. Since the test suites were written, regression testing was disciplined and proved to be valuable. Features that had worked for weeks stopped working as we moved from the developers' computer to the end-users' machine. It would not have been likely that these features would have been tested, if an ad hoc testing approach had been used.

The design notation reduced the typical learning curve one encounters when he/she moves from a procedural approach of solving problems towards an object-oriented approach.

The design notation was the common communications vehicle between programmers during walkthroughs, during debugging and as one programmer reused the code of the other programmer. The design notation was more effective at communicating how the program worked than the code.

The 3D-HIP documents simplified project tracking. The individual units identified in the model, along with their test suites, constituted project milestones. There was no question about what was in a work unit or when it was completed. We took another advantage of the clear distinction of units within 3D-HIP by implementing the areas with perceived high risk early in the project. Furthermore, functionally of the program was so clearly divided in the 3D-HIP documents that it was possible for us to deliver early, a partially working program to allow the end-users to begin entering their manufacturing processes and product line information.

The systems designer experienced difficulties in his personal life which made him effectively unavailable for 40% of the implementation effort. The documents and models were satisfactory and no impact to the project was felt due to his absence.

Summary

In this paper, we stressed that low defect software can be written for a reasonable price. But to achieve such results involves a commitment and focus towards continually improving ones software development process. It means taking the responsibility to define methodologies that fit the applications at hand. We presented one aspect of our process to illustrate that one can take ideas from several places and mold them into a combined methodology that is tailored to a specific need. This methodology may be effective for the reader's environment, but the more important message is that one has the power to not rely on some outside methodology guru, but rather to define their own methodology.

Roles of the Authors

This paper has been written using the word *we* to reflect the notion of team participation. But, we wanted to be clear about the roles each team member played. Norm is the person who has researched modeling techniques and developed the process discussed in this paper. Furthermore, he was the person who developed the information model, the event-partitioned data flow and the 3D-HIP documents of the Section 5 Scheduling Manager. Bob and John were added to the project once the specification was complete and they agreed to work within and develop the process further. Norm, Bob and John developed the design documents. Bob and John developed the code. Bob and Norm developed the test suites. Bob managed the project.

References

[Boehm] "Characteristics of Software Quality" by Barry W. Boehm, et. at., North-Holland, 1978.

[DeMarco] "Structured Analysis and System Specification", Tom DeMarco, Yourdon Press, 1978.

[Gause] "Exploring Requirements: Quality Before Design," by Donald C. Gause & Gerald M. Weinberg, Dorset House Publishing, 1989.

[Goldratt] "The Goal - A Process of Ongoing Improvement," by Eliyahu M. Goldratt and Jeff Cox, North River Press, 1986.

[Hatley] "Strategies for Real-Time Systems Specification," by Derek J. Hatley & Imtiaz A. Pirbhai, Dorset House Publishing, 1988.

[Humphrey] "Managing the Software Process," by Watts Humphrey, Addison-Wesley Publishing Company, 1989, pg. v.

[Kerth] "The Use of Multiple Specification Methodologies on a Single System," by Norman L. Kerth, Fourth International Workshop on Software Specification and Design, April 3-4, 1987, Monterey, California, sponsored by the IEEE Computer Society.

[McMenamin] "Essential Systems Analysis," by Stephen M. McMenamin & John F. Palmer, Yourdon Press, 1984.

[Shlaer] "Object-Oriented Systems Analysis - Modeling the World in Data," by Sally Shlaer & Stephen J. Mellor, Yourdon Press, 1988.

[Ward] "Structured Development for Real Time Systems," by Paul T. Ward & Stephen J. Mellor, Yourdon Press, 1985, 3 Vols.

[Weinberg] "Rethinkig Systems Analysis and Design," by Gerald M. Weinberg, Little, Brown and Company, 1982.

A Software Development Methodology: Fitting a Soft(ware) Peg into a Hard(ware) Hole

D. Barnes and J. Malsbury
Princeton Plasma Physics Laboratory
P.O. Box 451
Princeton, N.J. 08543

Abstract

Experiences when establishing a software development and change control program in an operational/research environment are discussed. The Princeton Plasma Physics Laboratory is a Department of Energy funded facility, primarily engaged in fusion energy research. Software quality assurance programs and principles were first introduced in 1981. The initial methodology was a formal and all encompassing program modeled after established hardware quality assurance programs. Software quality assurance has since evolved into a more flexible program, focusing on those systems that are more likely to effect overall reliability or safety. Details of both the early implementation and today's methodologies are discussed. Lessons learned and recommended practices are presented.

Biographical Sketches

D. Barnes is the head of the Computer Division at the Princeton Plasma Physics Laboratory (PPPL). She has been with PPPL over 13 years. Previous positions held at PPPL include scientific applications programmer, Group and Section head of Applications Programming and Branch Head of Computer Operations. She holds a B.A. in mathematics from the University of Colorado.

J. Malsbury is the head of Assurance Engineering at PPPL. She has been with the laboratory for 14 years. She served as a Software Systems Analyst in the Computer Division, before joining the Quality Assurance organization in 1983. Ms. Malsbury holds a B.S. in mathematics from Douglass College and an M.S. in Computer Science from Stevens Institute of Technology. She is also an American Society for Quality Control Certified Quality Engineer.

This work is funded by the Department of Energy under contract DE-AC02-76-CHO-3073.

Introduction

This paper discusses the evolution of a Software Quality Assurance Program at the Princeton Plasma Physics Laboratory. A brief overview is presented. Two aspects of software management are then discussed in detail; the System Development Process and Change Management. Lessons learned and recommended practices are presented.

Overview

Context

It is important to keep in mind the context in which one is working. Organization mission and existing management structure must play a part in the successful implementation or modification of any major program in any organization. For example, a software quality assurance program implemented at a bank (where security should be a major issue) might have very different components or emphases from one implemented on a nuclear submarine (where reliability should be a major issue), or a scientific research project (where the ability to do rapid prototyping may be required).

Princeton University's Princeton Plasma Physics Laboratory (PPPL) is a Department of Energy (DoE) funded facility dedicated to research and development of plasma devices. The ultimate goal is to develop a commercially feasible fusion reactor. Although the primary mission of the laboratory is research, the large size of our project dictates that many factors must be considered when piecing together successful system development procedures. Lack of emphasis in the areas of reliability and security, can affect a large number of related systems.

Some History

This energy research began at PPPL in 1951 under code name "Project Matterhorn" as a classified project. Fusion research became unclassified in 1958. Numerous plasma and fusion devices have been designed, built and operated at PPPL through the ensuing years. Construction on the Tokamak Fusion Test Reactor (TFTR) began in 1977. TFTR was designed to be the flagship of the United States fusion program. The mission of TFTR is to reach "scientific break-even". This occurs when the total fusion power produced is equal to the plasma-heating input power.

TFTR became operational on Christmas Eve, 1982. The construction cost of this device was originally estimated to have been $340 million with annual operating costs of about $60M. The construction cost of TFTR gives some indication of the complicated nature of the hardware involved. Obviously a quality assurance program was necessary.

TFTR was to be the first experimental device at PPPL to be computer controlled. The design of this control system, the Central Instrumentation, Control and Data Acquisition System (CICADA), began in 1976, in preparation for the 1982 operational date. The staffing profile shown below (Fig. 1.) provides some insight into both the size and complexity of the project.

Prior to 1981, there was no formal software quality assurance at the Laboratory. Individual groups developed systems for reviewing designs and testing products, but there were no formal written procedures or practices. DoE began expressing an interest in software quality assurance and requested that their subcontractors, such as PPPL, establish plans. During 1981-1982 individuals were assigned the responsibility of producing a computer systems quality assurance plan. The first formal Systems Development Procedures and Change Management Plans were adopted by 1982. The procedures and plans have continued to evolve as the mission and goals of the support organization, the Computer Division, have matured.

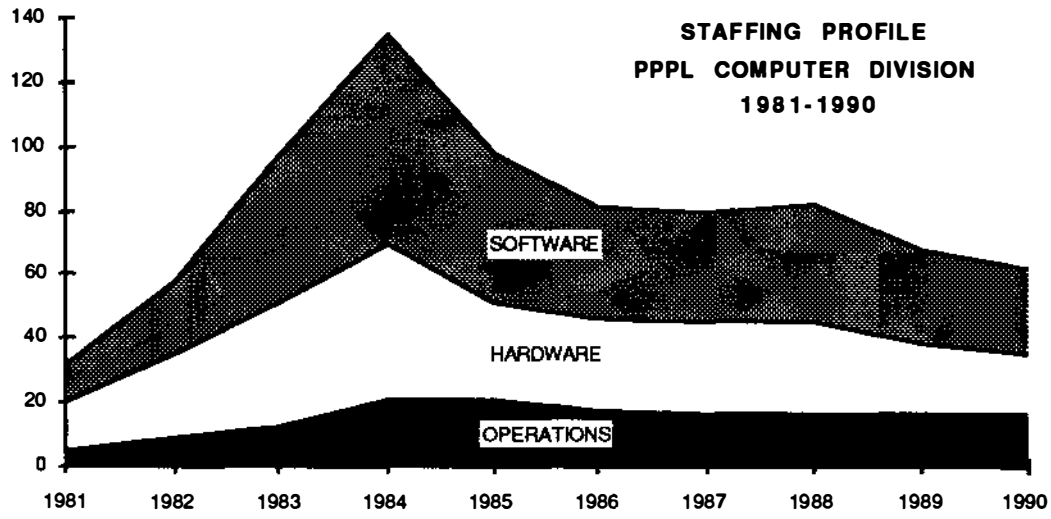


FIGURE 1

Several elements played an important role in establishing the scope and direction of procedures and plans. Given the complexity of the project, there really is no question that some quality assurance techniques should be applied. CICADA is comprised of systems that typically have both hardware and software components. In the early phases of the CICADA project, it was convenient to address a system rather than separate the hardware and software. With hardware and software elements combined from a project management viewpoint, the first Quality Assurance methodologies adopted were based on hardware quality assurance procedures and principles. At that time, there were well established hardware quality assurance procedures both within the laboratory and within the quality field.

The System Development Process

An overview and the scope of the System Development Process at PPPL is presented. Three aspects of Systems Development Methodologies are then addressed; 1) Project Management, 2) The Software Life Cycle, and 3) Documentation,Reviews and Approvals. Documentation and reviews take place throughout the Software Life Cycle.

Overview

From the conceptual design through the initial implementation of CICADA in 1982, extensive, formal and rigid system development methodology principles were proposed. We had to start somewhere, and a formal system, such as was in use for our hardware system design seemed appropriate. As experience was gained with the TFTR and CICADA projects as well as with the implementation of the system development procedures, we were able to streamline our System Development procedures to more closely match the environment at PPPL. This has had a positive effect on productivity and employee morale.

Scope

Key in the difference between the early approach and the current one is the introduction of the term "Critical". In the early approach, ALL software was subject to ALL phases of Software

Quality Assurance Procedures, including all Life Cycle Phases, the full documentation requirements, and the change management procedures. In the early phases of the development of the CICADA system, we were dealing with an unknown, so everything appeared to be important. As we gained experience, we were able to differentiate levels of risks. In the new approach, the procedures only apply to "Critical" software. Critical is defined as being essential to on-line operations, a high risk item, having the potential for creating a personnel safety hazard, or a product on which other products have many dependencies. All other software is "non-critical". For example, the data acquisition code is critical; the code controlling the amount of gas pumped into the reactor is critical; the code monitoring and displaying the trace element components in the gas is not critical. By making this Critical/Non-Critical distinction, 75% of the software products developed became exempt from the formal procedures. This allowed us to concentrate our efforts on those components that really effect overall reliability, leaving the quality assurance of the non-critical components up to the discretion of line management.

Project Management

A Task Force Approach, a matrix management system, was first implemented to manage the subprojects within the CICADA project. As hardware and software elements were combined in the definition of these subprojects, this matrix approach provided necessary communication paths. As the number of these subprojects multiplied however, it became increasingly difficult to define and enforce lines of responsibilities. An engineer might be a Task Force Leader on one project, a Task Force member on two or three other projects, and a project reviewer on two more. Scheduling and prioritizing work became an issue. We eventually reverted to defining responsibilities through the already existing line management structure of the organization. Although project/matrix management can be a correct approach, in our situation there were too many "projects" with too few staff to implement this approach effectively. The implementation of the project approach was further aggravated by the choice of projects. They were frequently defined as small (person-months of effort) rather than large (person-years).

In our current approach, the line managers meet together once weekly to status all projects. Additionally the line managers conduct staff meetings once weekly. The average engineer attends 1 regularly scheduled meeting each week, and an occasional Design Review. The average line manager attends 2 regularly scheduled staff meetings per week, project statusing meeting , a weekly Division Technical Review meeting and Design Reviews, as required. Required communication paths have been minimized; time spent in both formal and informal meetings has been minimized. There has been some attempt to define projects on a grander scale, however, due to the primarily maintenance phase of the overall system, many project definitions remain small. There is, an overall perception of increased productivity.

Software Life Cycle

The first Software Life Cycle methodology introduced consisted of seven rigid steps, as shown in Fig 2. Each step had to be successfully completed and all required documentation approved before the next step could begin. However, in reality products were sometimes on-line before approvals were obtained for the first step. The current Software Life Cycle has evolved to be 4 loosely identifiable phases. Major differences are the elimination of the Requirements Analysis as a formal phase, the combination of the Design Phases and the combination of the Testing Phases. These combinations simplify one's perception about the process.

Software Life Cycle: Product Identification and Approval

The major difference from the previous plan at this step in the life cycle process is that the Requirements Analysis Phase has disappeared as a formal element, replaced by the simpler "Product Identification and Approval" phase. Although it may seem unreasonable to many to embark on design without first defining requirements, we have found in practice that what our user/customers "require" is really a function of what is available, both in terms of resources and cost. Defining the requirements is something of an iterative process. We had spent way too much time in this phase as a formal requirement, perhaps because we were not trained in how to

really perform Requirements Analysis. For our application codes, it works best to get into the Design Phase as early as possible, with Conceptual and Preliminary Design Reviews up front to facilitate a constructive dialogue between the user and the developer. The Conceptual Design phase is where Requirements Analysis, as such, is accomplished. In general, like with our original approach to project management, our projects were just too small for rigorous and formal requirements analysis to be an effective tool.

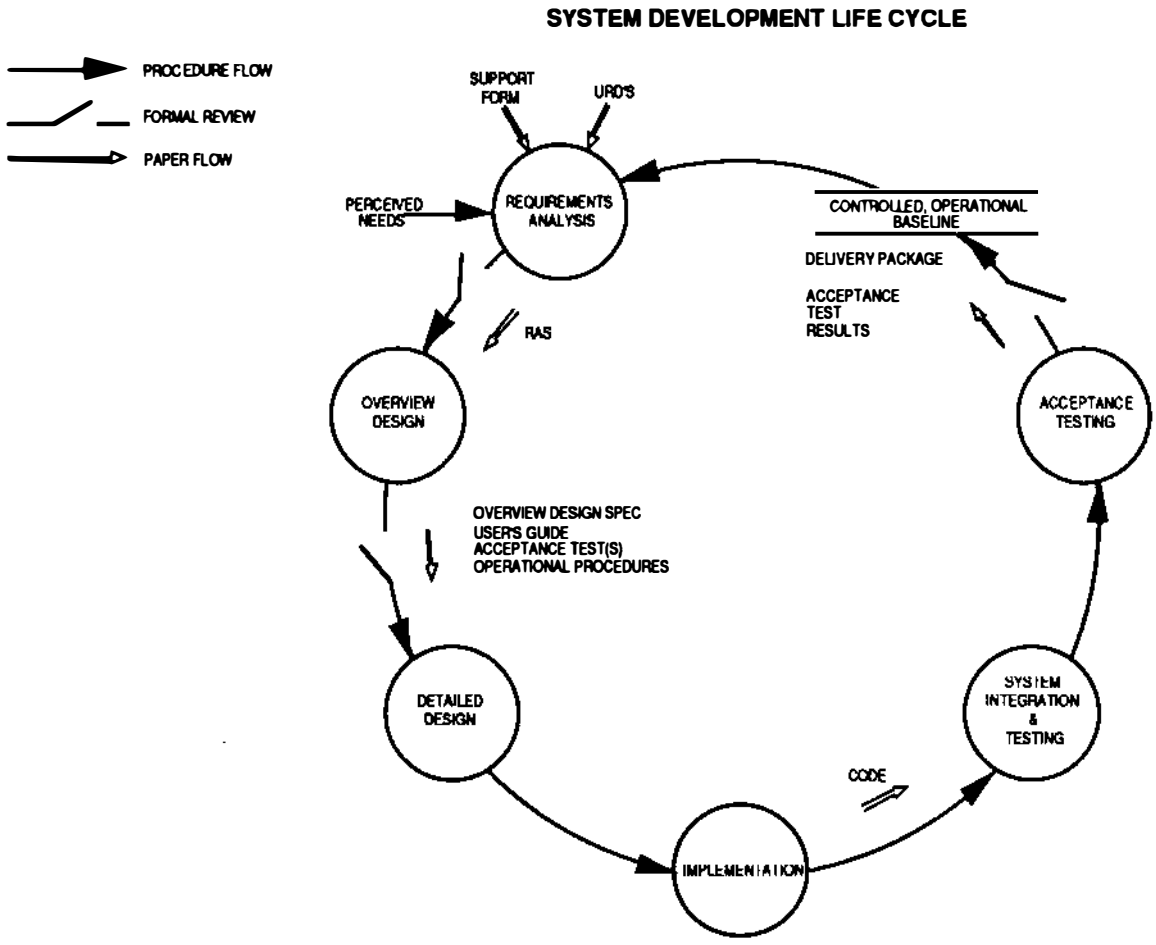


FIGURE 2

In our current implementation, the Product Identification and Approval phase simply describes how a user makes known the need for product development, enhancement or modification, and how this need is accepted and recorded into the system. Both technical and administrative approvals are required before work can begin on a product. Many aspects of this phase are specific to PPPL's Program Management System, and thus not relevant to this discussion. The technical approval process is, however, of importance in the Software Life Cycle.

The assignment of a product to a particular work unit, know as a Section, is made by the Deputy Division Head (see Fig.3.). The assigned Section Head determines product criticality. If a product is determined to be critical, the Section Head completes a one page, 2 sided, "Critical Job Form"(CJF). The form provides a brief description of the product, and the reason for criticality, but most importantly it defines the required approvals, reviews, membership on review panels and

required documentation for that product. Each critical product has an up front and individually tailored quality assurance program associated with it's development. For example, a code to perform data compression (critical to data integrity) might be defined to require 3 levels of design review, a design specification, a user's guide, operational procedures, a test plan and installation approvals, whereas a graphics library code (critical because of other product dependencies) may only require one design review, a design specification and a user's guide.

PPL COMPUTER DIVISION FUNCTIONAL ORGANIZATION
MAY 1990

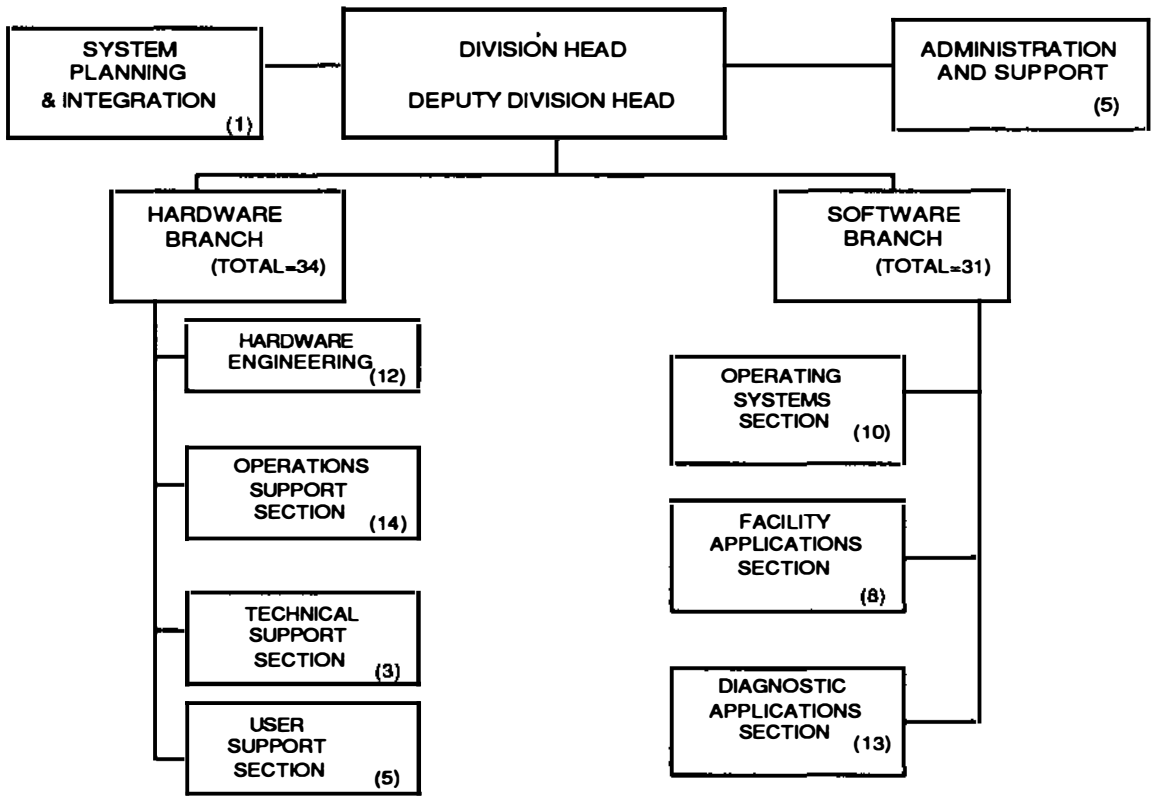


FIGURE 3

The Section Head, after completing the CJF, submits the form to the Division Technical Meeting (DTM) a weekly management meeting attended by the Section Heads, Branch Heads, the Division Head and a representative from Quality Assurance. The form is reviewed at this meeting. The appropriate Branch Head and Division Head provide final approval. We currently process less that 1 of these per month. Less that 20 are currently "open". Prior to the introduction of the criticality concept, and when the Task Force Approach was used, some 80-100 Task Forces existed, managing these small scale projects. We were therefore formally managing 80-100 subprojects. This effort is now not only concentrated on, but also tailored for the 20 or so most important efforts.

Software Life Cycle: The Design Phase

In the current methodology, two design phases have been combined. On paper, the earlier methodology appears to be more extensive and formal. Each design phase had to be completed before entering the next phase. Coding couldn't begin until both Overview and

Detailed Design phases were complete. In fact, coding had to, and did begin prior to the completion of the design, in order to estimate resource requirements, to test concepts, etc. At that time, formal design review standards did not exist, however that reviews take place and the exact reviewers were stringently specified. Many documents, from User's Guides to draft Acceptance Tests were required at these phases.

In the current methodology, the CJF can specify up to three levels of Design Review: a Conceptual Review, a Preliminary Review and a Final Review. Both the number of reviews and the reviewers are tailored to the product. Each review might have different participants and reviewers. Design review standards have since been developed for the TFTR project, which are now well understood, accepted and followed. The result is that reviews are conducted only when required, the reviews are attended by the appropriate personnel, and the reviews are conducted according to well established standards.

Software Life Cycle: Coding and Fabrication Phase

Known as the Implementation Phase in our earlier methodology, one interesting evolution in this phase is that we now code before all the design has been completed. This, of course, has encouraged prototyping where applicable. In practice we occasionally conduct the final design review months after a product is on-line and as a result accept a less than perfect design because the product already exists. This is a real risk. We, however, have found that the potential gain is worth the minimal risk. Most of the time, having the ability to code and prototype early has lead to a stronger, more clearly understood design of the final product.

We recognize a need for Coding Standards defining commentary or error reporting mechanisms, for example. Through the years, work groups have come to adopt particular implementation styles. For example due to organizational and historical considerations, we have, in existence, three distinct styles for Error Reporting . Clearly it would be better to have documented standards. Documenting and enforcing such standards, however, has not been a high priority or likable task.

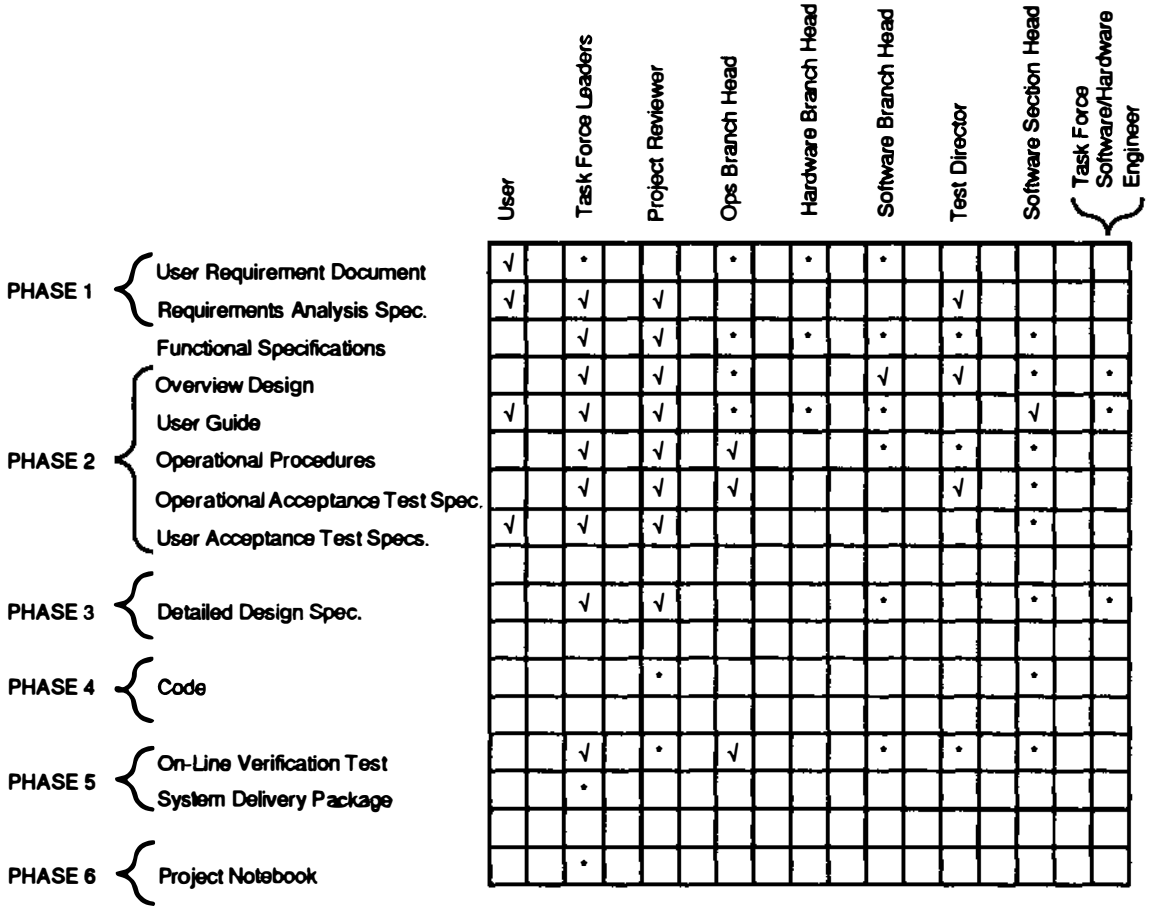
Software Life Cycle:Testing and Installation

Again, in our current implementation, two previous phases have been combined simplifying the formal testing processes. Much of the final testing and installation must take place on the on-line system, with the possibility of effecting reliability. Guidelines and procedures for testing and installing critical software have been formalized. These guidelines specify necessary steps to be followed. Standard test times are specified. Scheduling and coordination of testing is discussed. Monitoring of the testing and backout procedures are prescribed. A procedure exists for escalating conflicts between testers and operational personnel. In general, there had been confusion over exactly how to practically test a product on the same on-line system required for TFTR operations. Providing the practical "Who do I see?, Where do I go?" type of document and obtaining user agreement on the content, seems to have partially alleviated the inevitable battle that naturally (and healthily) exists between developers and operations personnel. Many of our products continue to exist in a developmental phase over long periods of time because of the "experimental" nature of the products, therefore, change control policies are applied to codes in this testing phase, as well as those codes that have become accepted as operational. These developmental codes can be volatile, impacting operations and overall reliability. Part of the testing is actually operating with these codes, so it is necessary to be aware of the changes and expected impacts.

Documentation, Reviews and Approvals

Figure 4. details the documentation and approval requirements that were first implemented in our software development process.

DOCUMENTATION APPROVAL MATRIX



√ Sign-Off

• Copies provided for comment before sign-off

FIGURE 4

In our current implementation, documentation requirements, reviews and approvals are all specified on the CJF and tailored to the particular Critical Product. On non-critical products, no reviews or documentation requirements formally exist. These are left to the discretion of the Section Head responsible for the product. In reality, at a minimum, a project notebook is compiled and maintained by the lead engineer. The maximum number of documents that might be required are:

Requirements Analysis Document (almost NEVER required)
Design Specification
User's Guide
Operational Procedure, and
Test Plan

Reviewers and approvers of these documents are individually specified per project, and average 3-5 per document. Thus the maximum number of reviewers required to approve documentation for a given product is 5 documents x 5 reviewers, or 25. The more typical product requires 4

documents (no requirements analysis) with 3-4 reviewers. This is less that half the number of reviewers of the documentation in the early plan. Additionally, the formal requirements only apply to the Critical products. This additionally reduced the overall number of documents and reviewers.

A summary comparison of documentation and review requirements is:

	Early System	Today
# of documents	13	5
# of approvals	29-61	5-15
# of reviews	10	3

Summary Comments on System Development

In our earlier attempts, although the system looked good on paper, in reality we found that we had implemented more detail than was warranted for the scope and mission of the project. Particularly as the project moved to a more operational and maintenance mode, the extensive documentation, review and approval requirements became cumbersome. As the steps became too cumbersome, there was a tendency to by-pass ALL quality assurance. Three important evolutions to the System Development Procedures contributed to the success of the process today.

First is the concept of "Criticality". Having tried to apply a system to all products resulted in it not being applied well to many products. Restricting the application of formal procedures to the products that effect system reliability or safety, has resulted in those fewer important products being well reviewed and monitored.

Second, the entire system is streamlined and individually tailored to the product. Reviews and documentation are not conducted or produced when not necessary. The Life Cycle approach is still followed, but the phases are compressed. The burden of the Life Cycle is realistic rather than unbearable.

Finally, and probably most importantly, the responsibility for all software quality assurance for all products has been placed with the line managers. There is accountability.

Change Management

It is important to note that an off-line facility for unit testing software exists, but the on-line system is required for integrating software/hardware products or for integrating software into the CICADA system. All products have to be tested on-line. The on-line system is operational 5-6 days per week, 18 hours per day, leaving Sunday and graveyard shift available for non-interference testing. It is not practical to restrict testing to those times, so change management is necessary for codes in the Testing and Integration Phase of the Life Cycle. The approaches we took toward solving this problem are presented, followed by our implementation details and some conclusions.

Approach: Control vs. Management

In our early attempts to control changes to the on-line system, we directly applied our hardware management control processes to software. We attempted to control additions, modifications and replacements to a baseline, at a time when no tools were available for source control. It was discovered quickly that developmental software couldn't be managed in quite so stringent a manner. In this original system we had attempted to control all changes to the on-line system through a paper process, review boards and organizational entities. Forms were submitted requesting approval for changes to codes; a management board reviewed these

requests. A "Test Director" was responsible for coordinating all on-line testing. A "Duty Chief" was responsible for maintaining the integrity of the on-line system at any given time. As the on-line system grew in size and complexity, the change control system became untenable. There were too many changes taking place to continue to control in this fashion. Shortly before implementing a new process, the management review board was meeting 3 days per week, in three hour sessions, to review more than 1300 open change requests. We replaced the notion of CONTROL with the concept of MANAGEMENT and completely revamped our process for managing change on the on-line system.

Implementation

The change management system was implemented through various administrative procedures. Our implementations for scheduling and tracking changes (Rollover Meeting), establishing daily accountability (the Duty Chief) and defining the scope (the Critical Component List) are discussed below.

Rollover Meeting

The new system placed the burden of responsible change management on the first line managers. The management review board, which had been comprised of Branch Heads and Division Managers, was replace by a weekly "Rollover" meeting. This meeting is attended by the Section Heads. The goal of this meeting is to schedule all expected activities on the on-line system. This goal of scheduling, rather than controlling, produces a "can do" rather that "can't do" attitude at all levels in the organization. The meeting produces necessary communications at the level of the implementing individuals. Minutes of the meeting serve as the documentation for change approval. Additionally, a list of all components that have changed in the last week is distributed and discussed. Peer pressure provides the "check" to assure that activities that are in fact being scheduled and performed at appropriate times.

Duty Chief Concept

Additionally, we played a bit of a trick on the responsible first level managers, the Section Heads. We retained the "Duty Chief" concept, a designated individual responsible for the overall integrity of the on-line system at any given time. We, however, rotated the assignment of Duty Chief to the Section Heads and some of their senior engineers. These first line managers now were not only accountable for the integrity of the products produced by their people, but also for the integrity of the entire on-line system through their participation in the Rollover meetings and their periodic assignment as Duty Chief. We made the "robbers" who had devised ways to "get around the system" into the cops, responsible for the integrity of the system. It worked.

Critical Component List

Rather than attempting to manage all software, we defined those modules that had the highest probable impact on system reliability or safety, and singled these items out to be better managed. A living list of these "Critical Components" is maintained by Operations. In general, "Critical Products" from the System Development Process produce the "Critical Components", although this is not a rule. Section Heads are responsible for defining those items to be placed on the list. The list defines responsibilities and authorities for installing critical components into the on-line baseline. An important aspect of our implementation of this process is that we do not require written approvals. Installation approvals are verbal. This implementation was a direct result of our reaction to an overload of paperwork and required signatures in the early system.

Some Comments/Conclusions on Change Management

Although our first implementation of a change management program was viable initially, as the system became larger and more complex and as the staff level increased, the system became too cumbersome to be practical. The lack of a true off-line facility for final software integration and testing further exasperated the problem.

The current system emphasizes communication rather than control. Little formal paperwork is involved. Installation approvals are verbal; scheduling and testing approvals are done in a round table fashion. The minutes of the Rollover meetings and entries into the Operators Log Book at the time of an installation are the only records of transactions. Although this makes auditing the processes difficult, daily life is simpler. Accountability is not lost. The line management of the developers has been given the additional responsibility for daily operational reliability. The CICADA system is now more mature so overall system reliability may not be a truly accurate measure of success of the change management program. However, reliability has risen from 89% to 96% in the past 4 years.

Summary

An evolution of the Software Quality Assurance Program at PPPL have been presented. Early efforts took established hardware quality assurance principles and procedures and directly applied them to software. As the size, scope and mission of the organization grew it became clear that we had been concentrating our efforts on small pieces, rather than the whole. As a result, the more "critical" aspects of our system were getting as much (or as little) attention as the less significant products. Overall system reliability decreased, as developers avoided rather than participated in software quality assurance.

The current methodologies evolved into a more relaxed and streamlined approach. "Critical" systems are identified and proper reviews and documentation are generated for these most important products. Quality assurance requirements for the non-critical products are left to the discretion of the line managers. Those systems that have a higher probability for effecting overall system reliability do receive proper attention and there is direct accountability. The result has been a higher overall system reliability, higher productivity and higher morale.

Lessons Learned

Throughout this paper there are several statements and implications of lessons we learned from our early attempt and are learning today. A summary of a few of the more important lessons on what to do right is:

1) **Involve the software developers**; forcing seemingly inappropriate rules on professionals creates morale problems.

2) **Concentrate on what is really important**; don't dilute your efforts by attempting to manage everything.

3) **Define what you are really trying to achieve**; the mission of the organization should drive the goals of a successful software quality assurance program.

4) **Accountability** is necessary.

5) Give the developers the broader **responsibility** for overall system reliability.

6) **Allow the methodology to be a living organism**. As the organization that the methodologies support evolves, so should the methodologies. Don't be tempted to continue use of old practices too long, just because they exist. Review your formal system annually for compatibility with your mission.

Continuing Activities

Areas where we believe there is still room for improvement include:

1) We still don't have formal standards. Although "styles" exist in practice, they are not formally documented. Actions have been assigned to staff members to produce these formal documents, but schedule pressures and constraints continue to interfere. An incorrect solution would be to adopt "commercial" standards, or assign producing the standards to uninvolved personnel. It is essential that the developers document their own standards. We are better off living without the formal documents than creating or adopting ineffective ones.

2) The flexibility in the life cycle process can allow products to be on-line before design criteria are met, resulting in unreliable or expensive to fix software. Direct accountability alleviates this tendency. We have found that having the flexibility for prototyping outweighs the potential problems. Control is exercised either through peer pressure, or in extreme cases through managerial actions.

3) Use of CASE tools is a continuing issue. We continue to evaluate CASE products at a low level, but have not yet identified products that are efficient for our environment. We are currently in an operational phase, modifying and enhancing products to fit within the existing structure. If a new major system design effort is initiated, our evaluation efforts may receive a higher priority.

4) Tractability remains an issue. It is difficult for the Quality Assurance Engineers to ascertain adherence to our procedures and to measure the effectiveness of our system in order to help us devise even better ways of doing business.

5) Metrics do not exist. We intuitively believe that we have evolved to a better system for our current needs. Our reliability has increased, the software engineers are happier and our customers are happier. However, without a way to measure the effectiveness of the system, it is difficult to fine tune or enhance the system.

6) Training always has been and remains an important issue. We continue to train our engineers in the state of the art tools of their trade like languages and operating systems, but training in the latest techniques for software development is a less popular item.

A Final Thought

If you have an environment similar to the environment at PPPL, you may be tempted to implement our current methodologies. DON'T. Learn from a few of our false starts and successes, and implement a methodology, perhaps founded on some of our principles, but designed by your own staff.

To be successful a software quality assurance program should be a natural and normal way of doing business for the people actually doing the business.

CONCEPTUAL OBJECT-ORIENTED DESIGN

Mark A. Whiting
Pacific Northwest Laboratory
PO Box 999
Richland, Washington 99352

ABSTRACT

Conceptual object-oriented design (COOD) is a methodology that is being used at the Pacific Northwest Laboratory (PNL) to study, plan, specify, and document high-level solutions to large-scale information processing problems. COOD embodies aspects of object-oriented program design philosophy (which is being applied to the implementation design of software) to provide enhanced tools and techniques for conceptual design. COOD is targeted at the phase of software development following requirements analysis and prior to implementation or detailed design. This step is necessary, particularly for large-scale information processing systems to achieve the following:

1. allow designers to conceptually work out solutions to information processing problems where innovative thinking is required,
2. allow a structured environment in which to capture design products, and
3. provide a global view of the conceptual solution in an understandable form to the implementors of the solution. This will facilitate their detailed design efforts.

The product of COOD is a "conceptual design specification." This specification is delivered to an implementation team to assist the detailed design process, yet is not a software specification in and of itself.

AUTHOR BIOGRAPHY

Mark A. Whiting is the Technical Group Leader of the Knowledge-Based Systems Group at Pacific Northwest Laboratories in Richland, Washington. His research interests include software engineering, programming languages, and hypermedia systems. He has been involved in the study of programming languages and compiler design for the past 14 years and designed and built his own object-oriented language in 1983.

E-mail address: `whiting%snuffy@pnlg.pnl.gov`

CONCEPTUAL OBJECT-ORIENTED DESIGN [1]

Mark A. Whiting
Pacific Northwest Laboratory
PO Box 999
Richland, Washington 99352

ABSTRACT

Conceptual object-oriented design (COOD) is a methodology that is being used at the Pacific Northwest Laboratory (PNL) to study, plan, specify, and document high-level solutions to large-scale information processing problems. COOD embodies aspects of object-oriented program design philosophy (which is being applied to the implementation design of software) to provide enhanced tools and techniques for conceptual design. COOD is targeted at the phase of software development following requirements analysis and prior to implementation or detailed design. This step is necessary, particularly for large-scale information processing systems to achieve the following:

1. allow designers to conceptually work out solutions to information processing problems where innovative thinking is required,
2. allow a structured environment in which to capture design products, and
3. provide a global view of the conceptual solution in an understandable form to the implementors of the solution. This will facilitate their detailed design efforts.

The product of COOD is a "conceptual design specification." This specification is delivered to an implementation team to assist the detailed design process, yet is not a software specification in and of itself.

INTRODUCTION

In the the software engineering lifecycle used at PNL, the design phase is separated into two segments. The first is one of conceptual design, where the solution space of the problem as a whole is developed. The second is that of detailed design, where an implementation description is derived. This separation has proven particularly useful for large-scale problems requiring a great amount of conceptualizing by multi-person teams. The following paragraphs describe conceptual object-oriented design (COOD), a top-down approach currently in use at PNL for high-level conceptual design and concept evaluation.

The COOD approach consists of a preparatory stage followed by the core conceptual design. The preparatory stage identifies the focus of the design task and then generates an informal specification to solve the problem. The following stage derives the object-oriented blueprint. The essence of the design method is based on iterative object decomposition, which roughly corresponds to functional decomposition in traditional design. This approach draws upon Booch's approach for designing Ada software (1983). We have modified the approach in consideration of the two-segment design process and the review and documentation requirements of PNL projects. The complete process consists of the following steps:

1. Preparation and Goal Setting
2. Requirement/Problem Review and High-Level Solution
3. Object-Oriented Blueprint
   3.1 Identify top-level conceptual objects in the solution space
   3.2 Identify object relationships and object dependencies
   3.3 Test solution with scenarios
   3.4 Descend into conceptual design; iterate to 3.1 until level of design has reached appropriate depth
4. Handoff to Detailed Design Team.

Pacific Northwest Laboratory currently uses COOD to design solutions to information processing problems being addressed by the Vertical Integration of Science, Technology and Applications (VISTA) project. VISTA is a long-term research effort conducted by PNL directed toward assisting in the clean-up of the nation's hazardous waste problem. The initial goal of the program is to develop a software-based information system to guide the assessment and remediation process for hazardous waste sites at the U.S. Department of Energy (DOE) facilities. The proposed Environmental Restoration Information System (ERIS) will link users (DOE, national laboratories, and remediation contractors) to computer models and technical data available at PNL, to speed up the remediation process, while decreasing costs and accelerating the deployment of new technologies. We have successfully designed and built components of ERIS using COOD, as integrated into an overall software development strategy. A description of the conceptual design approach follows, with simple examples taken from the ERIS design. The context for this example is the design of a tool to assist users in organizing and performing complex procedures that are to be carried out on a computer or with the aid of a computer--essentially, an electronic laboratory notebook.

PREPARATION AND GOAL SETTING (Step 1)

Decide on the bounds of the design process to be performed. Conceptual design should be thorough, yet the process may not always need to be completed in one design session. A particular task or project may require a specific subset of the design process to be performed to verify problem understanding or solution concepts or to sell your solution to a customer. This step arrives at agreed upon bounds and establishes that there can be an end to the process.

REQUIREMENT/PROBLEM REVIEW AND HIGH-LEVEL SOLUTION (Step 2)

Generate a short statement of the problem (Figure 1). A paragraph can serve as the

Figure 1. Problem Statement.

At this point, the design team had come to an important issue in the design. The component that a user would be focusing on to complete a subtask needed to be considered. After some discussion, the concept was succinctly captured in the problem statement.

"We need something that will give us a focal point for the task/application to be run -- it should automatically provide guidance, allow execution, maintain a record of the process runs and results, and allow us to record observations for auditing purposes."

focal point for the design team as to the requirement they are addressing or the problem to be solved.

Determine what will solve the problem (Figure 2). Brainstorm solutions. The team must, however, arrive at a theme for the solution concept so that an object-oriented blueprint can be created. It should be written down or otherwise captured so that subsequent steps do not become overly digressive.

Figure 2. High-Level Solution.
The high-level solution also required much discussion, and its essense is captured here. Note, we still attempt to maintain an even level of abstraction, even though we are in the midst of the object decomposition:

"A topic is the basic entity of the lab notebook. It contains guidance and the executable specification of how a process is carried out, maintains an automated history and allows user annotation."

OBJECT-ORIENTED BLUEPRINT (Step 3)

Perform a top-down, component-wise object decomposition of the solution. Ultimately, a thorough conceptual design specification that is understandable to an implementation team will be generated. Products constituting the specification include: object design templates, object-relationship diagrams, object interface diagrams, user view depictions, and design tests via scenario descriptions. The process is both iterative and recursive. It is iterative in the sense that the team performs the same decomposition design steps across the breadth of the top-level objects identified in the solution space. It is recursive in manner of decomposing design objects.

Identify Top-Level Conceptual Objects in the Solution Space (Step 3.1)

Identify the key conceptual entities of the system and identify the operations associated with each entity (Figure 3). Unfortunately, there are no formal methods for identifying objects and operations in a solution space. The method advanced by Booch involves extracting the nouns and noun phrases from his informal solution strategy paragraph. Discarding irrelevancies, this would serve as the starting point of an object list. Identification of verbs and verb phrases would provide the initial set of operations on the objects. Attributes of the nouns and verbs may be identified as object characteristics (e.g., a "blue" thing), action characteristics (e.g., a "fast" activity), or a component part of the object. Our general feeling about this method is that it may provide a good starting point, but unless you are very confident about your solution paragraph, you may not generate a satisfactory list. The process used by Beck and Cunningham (1989) in designing software using CRC cards is essentially a scenario-based analysis of the solution space. They lead a person through a design, suggesting cases that may contribute objects or change the responsibilities of objects in the design. It is an exploratory process that results in a good, usable set of objects, if you have done an acceptable job in exploring the cases to which the solution will be applied. Rosson and Gold (1989) concentrated on the "early phases of design" in observing experts performing object-oriented design on a selected problem. In the conceptual mapping from problem space to solution space, the experts' methods also seemed to be somewhat exploratory. As we designed innovative solutions to problems, an exploratory approach seemed both helpful and appropriate.

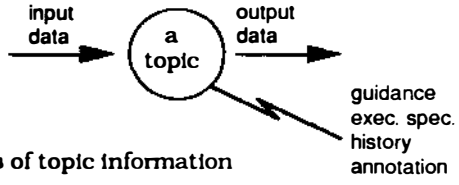
VISTA Object Design Template

"Topic"

Purpose:
A topic is the basic entity of the lab notebook. It contains guidance and the executable specification of how a process is carried out, maintains an automated history, and allows user annotation.

Characteristics:
A topic can be viewed similarly to a process node in a data flow diagram--the node representing a process and information flowing from children nodes to the topic to its parent nodes. The topic itself is a composite containing guidance, history, and so forth.

Illustration:



Responsibilities:
provides guidance
manages I/O
provides various views of topic information
executes itself
records information about its execution
allows the user to record information about its execution

Collaborations:
Guidance/help
History
Annotation
Script

Unresolved Issues:
- How/when does the user trigger execution of the topic?

Figure 3. Object Design Templates.

In decomposing "topic", much of the design information that was discussed in the design sessions needed to be captured outside of the somewhat unstructured meeting minutes. The object design templates served this purpose nicely. As the information was being discussed, a member of the design team captured the information on a white board. It was later cleaned up and entered via object design templates into our computer-based documentation set. By the end of the conceptual design session, we had several pages of design templates for the laboratory notebook.

Whether using the Booch method or some other more exploratory approach, information collected about identified objects is consolidated into *object design templates*, more fully described in the tools section below.

Identify Object Relationships and Object Dependencies (Step 3.2)

Construct object-relationship diagrams to provide overview information about the solution (Figure 4). Construct object-dependency diagrams to show the interfaces and interactions between objects (Figure 5). A fundamental tool for considering the objects in the solution space is an object-relationship diagram. The object-relationship diagram is based on the entity-relationship diagram (Chen 1977) where the objects replacing the entities represent much more powerful system components. This diagram still specifies *what* constitutes the solution space and not *how* the solution functions. An object, specified as part of the solution, plus its compo-

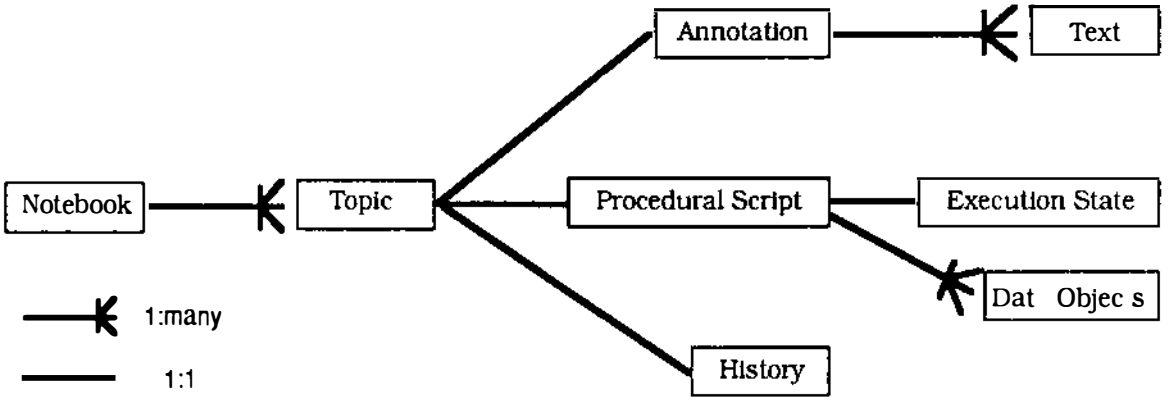


Figure 4. Object-Relationship Diagram.
As the object decomposition of "topic" proceeded, it became clear that a diagram capturing an overview of the component relationships was needed. The relationship between topic and other components of the lab notebook was captured in an object-relationship diagram.

nent parts plus each of their component parts and so on is an example of a useful structure for an object-relationship diagram.

The innovative solution will call out objects that depend upon other objects to accomplish their operations. Also, the designers may discover such relationships, as objects are being designed and recorded, that are not explicitly called out. The relationships between these objects should be captured in a more obvious fashion than through an informal description. A Booch interface diagram may be used to graphically depict object interfaces plus dependencies between objects. Another tool for capturing objects and dependencies is the CRC (Class, Responsibility, Collaboration) card. As mentioned above, the CRC cards have been presented as a tool to facilitate learning of object-oriented techniques. As a design tool, one shortcoming of the cards is that dependencies are not visually obvious when viewing the solution as a whole. The VISTA Object-Oriented Design tOOl (VOODOO) is a COOD CASE tool that addresses this problem. VOODOO captures both object data and object-relationship information, and is described in more detail below.

Note that a component-based object-relationship diagram, as mentioned above, consists of objects drawn from more than one level of this design process. It may prove more beneficial to the design team to construct one or both of these diagrams at certain points after multiple levels of the object design have been specified. They may then be augmented as the design proceeds.

Test Solution with Scenarios (Step 3.3)

Construct scenarios that reflect various aspects of the requirements specification. Test the design by walking through different scenarios that test the design boundaries--these may be best generated by people outside of the design team. Undoubtedly, partial scenarios or test cases

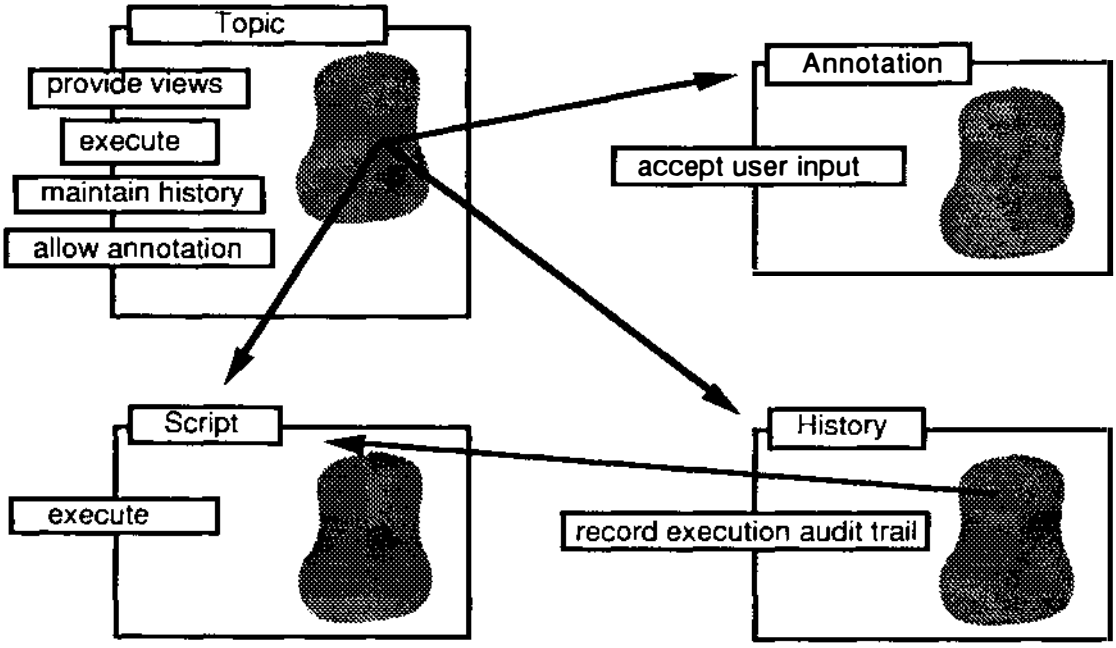


Figure 5. Booch Interface Diagram.

The other vital set of relationships that needed to be captured was that between an object and those objects it depends upon to complete its tasks. Again, by the end of the conceptual design session, several pages of interface diagrams had been generated.

are part of the continuing design as the team reasons about the responsibilities of objects. Obtaining a scenario from an external source will result in added understanding of the system and component interactions.

Iterate Until Level of Design Has Reached Appropriate Depth (Step 3.4)

Continue to identify objects at the next lower conceptual level of the design. At this point, the process recurses into the solution design space, where objects that are part of a higher-level composite are examined. The team chooses one and reviews it as described above. Similarly, objects at the same level of abstraction need to be iteratively reviewed and decomposed. The goal is to cover the breadth and depth of the solution space, while bearing in mind the goals of the process originally established.

An important question from the "goal setting" step is: what is a fundamental component of your conceptual design? This must be determined so that you have a specified stopping point. If the team has reached a level where objects are non-decomposable, this is a good stopping point. If the team finds itself looking at "objects" that don't do anything, e.g., contributing data elements that are not objects, this also is a good stopping point. The team may also reach a point where the team members, the users, and the reviewers all feel comfortable that the design has been adequately captured and that the scenarios work appropriately. This, again, may be time to stop the process and move on the next tasks.

Testing of the solution as a whole is necessary when the design is completed. "What-if"

VISTA Object Design Template

An object design template may be used to describe conceptual entities that represent objects for a software system. The templates should be filled in with information about the objects, its reason for existence, its function, perhaps an illustration if it helps to convey the central idea behind the object, its dependencies on other objects needed to do its jobs, and any outstanding unresolved issues the design team had after discussion of the object.

"Object Name"

Purpose:
This section includes a general statement about what the object is and why it is.
Characteristics:
Any additional information about the object that will help describe it is captured here.
Illustration:
A sketch of the object, if appropriate, is included here.
Functions/Responsibilities:
This section would be a list of the functions the object performs in the conceptual solution. Syntactic or semantic treadmills should be avoided while describing these--those discussions will take place at the implementation design level.
Dependencies/Collaborations:
A listof the objects this object depends upon to perform its job.
Unresolved Issues:
This section holds unresolved issues concerning the object that are raised during review sessions. Leave enough space so that issue resolution information may be incorporated at a later date.

Figure 6. Object Design Template Description.

scenarios may be exercised with the potential users, using the object design templates and other artifacts of the design in a similar manner as you would exercise an on-line prototype.

HAND OFF TO DETAILED DESIGN TEAM (Step 4)

At this point, the conceptual design can serve as input to the system implementors. However, the implementors need to perform detailed design (now emphasizing how the implementation will work rather than what the solution accomplishes) to generate the implementation description. A thorough description of this process is beyond the scope of this paper, yet it should be performed in a manner fundamentally different from the conceptual design described here. For object-oriented development, implementors will naturally want to take advantage of the software reusability and existing inheritance structures found in object-oriented development systems. This is a good argument for using the bottom-up approach for detailed design. Meyer (1988) discusses bottom-up strategies for software construction using the object-oriented language, Eiffel.

TOOLS FOR CONCEPTUAL OBJECT-ORIENTED DESIGN

We have developed the "object design template" (Figure 6) at PNL to be used in high-

level object definitions. The object template includes the object name, a short paragraph description of the object's purpose within the system (why it is present), a description of the object's characteristics (general object behavior and attributes), a space for illustrations, a section for a more detailed description of the object's functions or responsibilities (what it does), a section for dependencies or collaborations (who it needs to perform its functions), and a section for unresolved issues (what might be wrong).

VOODOO (Figure 7) is a Supercard™-based COOD CASE tool that allows the capture of individual object information, plus relationships or dependencies between objects. It is a composite tool, where object data is recorded in an object design template window, and dependencies are diagrammed in a box and connectors display. Multiple object design templates and

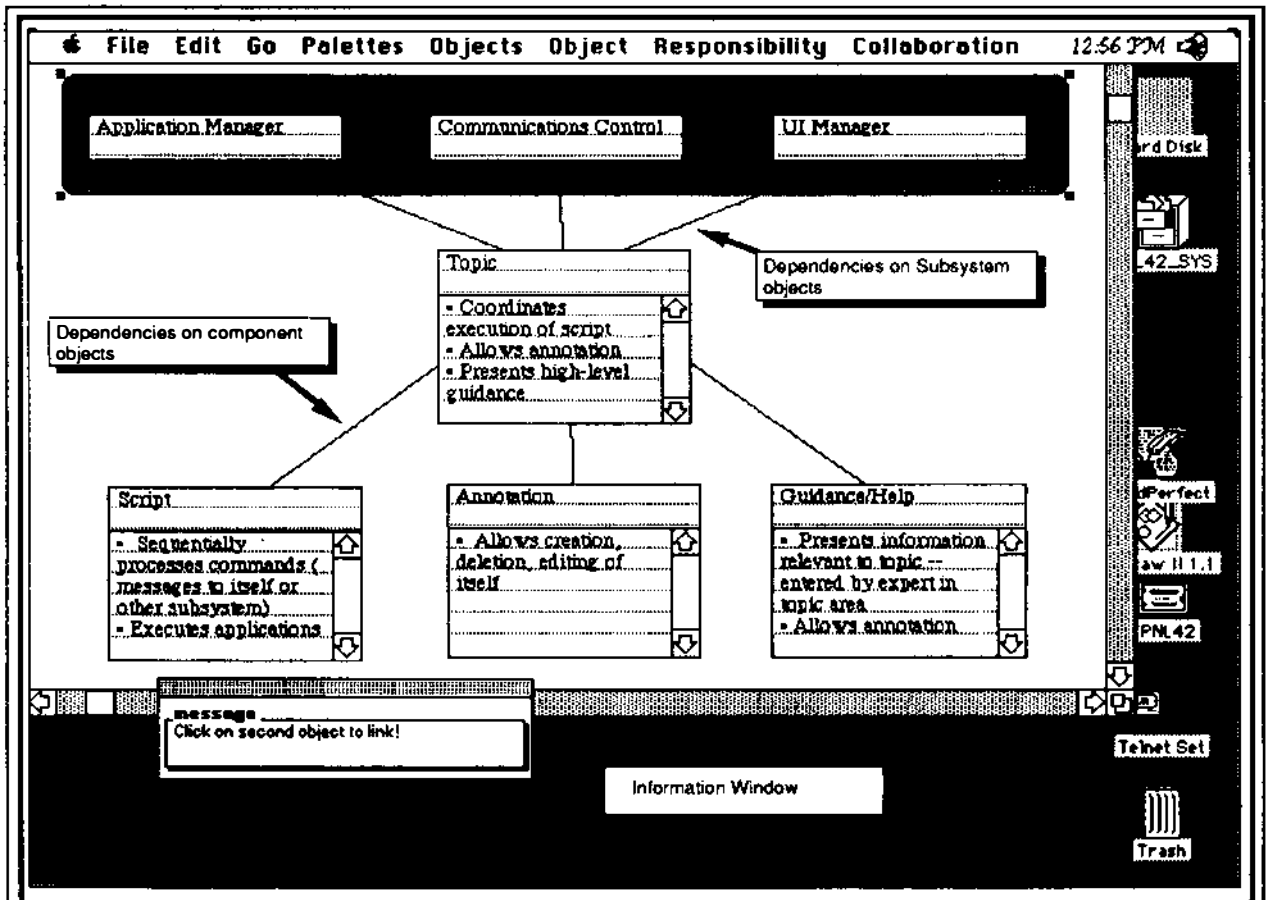


Figure 7. VOODOO, shown here executing under multi-finder on the Macintosh®, displays an object-relationship diagram involving Topic and cooperating objects. In this view, each object has a name field and a scrolling responsibilities field. Connectors between objects indicate dependencies. The connections directed upward show dependencies on objects outside of the electronic notebook subsystem. Here, it was sufficent to indicate those dependencies in a broad sense, individual subsystem objects had not been defined at the time. Also, the responsibility sets for the subsystems did not have to be entered (VOODOO allows that field to be hidden). Connectors directed downward indicate dependencies on objects within the notebook subsystem. Shadowed boxes were added to the diagram after it was initially built, but annotations to the diagrams can occur on the fly using SuperCard's text and drawing tools. The "message" box below is a floating palette that pops up with instructions or notes for the VOODOO user.

Note that the graphical view does not provide as much information about the objects as the user may want to know. The complement to the graph view is a SuperCard-based implementation of object design templates (not shown here). Mousing on an object would bring the user to the corresponding object template with more detailed descriptions of its purpose, functions, and interfaces.

diagrams may be created, edited, and stored in VOODOO. Diagrams may be printed according to users needs and object design reports may be generated in different formats. An advantage of VOODOO is that it captures the design history. Revised descriptions or diagrams are easily created with previous versions automatically stored within the Supercard™ project.

Conventional tools are often useful in an object-oriented design. A state-transition diagram, depicting the potential life-cycle of an object, is often helpful in clarifying a conceptual component of the design. The conventional format of the diagram, with nodes representing the states of an object, arcs representing the transitions the object proceeds through, and some specification of the events the object is responding to, is adequate to convey this information. A modification used in the VISTA project is to depict state-transition diagrams in HyperCard® or SuperCard™. States are drawn up on a card, events are programmed into buttons on the card. Mousing on a button implies occurrence of an event, arriving at the resulting card represents a new state of the object. The animation of the state-transition automaton is highly effective in conveying information about an object's life-cycle between designers. It should be remembered that this still represents a conceptual design tool, and the information to be conveyed is a conceptual depiction. Still, the HyperCard® or SuperCard™ animation could be helpful later to implementors who will be looking at this information with software construction in mind.

User views are potential user interface presentations that are a first cut mapping of the designer's model to a user's model. The tools used by designers (such as some of those described above) are not always easily understood by users. In checking that the designer's model of the system still corresponds with the user's vision, a different presentation medium can be used. User views usually represent a form of paper prototyping of the design, although HyperCard®, SuperCard™, and other prototyping tools allow quick, high-quality, automated construction of design concepts. The user views should not presume to be well-thought out user interface specifications, rather should be introduced as communication tools. If a user takes a particular liking to a format of the presentation, this can be taken as a fortuitous by-product to be carried into subsequent phases, but should not be a goal of the design team in constructing user views. In the VISTA development efforts, user views were used both before and after the COOD process, and were very effective in verifying concepts. In our case, the early user view was developed using Supercard, the subsequent user view using the Interface Builder™ on the NeXT machine.

CONCLUSION

Conceptual object-oriented design allows initial, innovative and creative design to occur via essentially a paper-based, object-oriented prototyping approach. The object paradigm provides a powerful base for organizing and capturing the design process and results in a form understandable to the design team and subsequent users of the design information. The results may even be used to construct on-line prototypes, where concepts may be examined and exercised more fully. The primary innovations in COOD are assistance in the area of the software development process to which the object oriented paradigm is being applied (early design), innovations and extensions to the Booch software design process, and in the creation of new and revised CASE tools.

COOD does not run against current thinking on object-oriented design as a whole in using a top-down approach. For example, although Meyer (1988) strongly advocates bottom-up

approaches, this is for implementation design, where reuse, extension and combination of objects are vital. This is exactly the process that needs to occur following the conceptual design stage.

REFERENCES

Beck, K. and Cunningham, W. 1989. "A Laboratory for Teaching Object-Oriented Thinking." *OOPSLA '89 Conference Proceedings*. ACM Press, New York.

Booch, G. 1983. *Software Engineering with ADA*. Benjamin/Cummings Publishing Company, Menlo Park, California.

Chen, P. 1977. *The Entity-Relationship Approach to Structured Design*. Q.E.D. Information Sciences, Wellesley, Massachusetts.

Rosson, M. and Gold, E. 1989. "Problem-Solution Mapping in Object-Oriented Design." *OOPSLA '89 Conference Proceedings*. ACM Press, New York.

(1) SuperCard is a trademark of Silicon Beach Software.
HyperCard and Macintosh are registered trademarks of Apple Computer.
Interface Builder is a trademark of NeXT, Inc.

This work was supported by the United States Department of Energy under contract DE-AC06-76RLO 1830.

Quality Software Prototyping through Object-Oriented Iterative Development

Daniel B. Leifker
The MITRE Corporation

Abstract

Software systems of the highest quality are often discarded because they do not satisfy the genuine needs of users. Many systems analysts insist on using "one-pass" software development strategies despite growing evidence that many software systems should be developed iteratively through thoughtful prototyping. Although prototyping can force elusive user requirements to the surface quickly, it is often resisted because comprehensive prototypes cannot be easily managed. Furthermore, users are tempted to convert successful prototypes directly into productional systems. An object-oriented framework, however, can bring enormous control to prototype management, and it can yield successful prototypes that need not be discarded once the user requirements are defined. This paper proposes a software development framework known as object-oriented iterative development (OOID). This framework is not necessarily limited to object-oriented databases; certain aspects of OOID readily support the development of conventional systems and can be used whenever prototype management and iterative development techniques are appropriate.

Author

The author, a member of the technical staff for The MITRE Corporation, supports the Mission Operations Directorate of NASA's Johnson Space Center in Houston, Texas. His present work involves the design of complex NASA systems that resist traditional methods of systems analysis. He has a B.S. and an M.S. in Computer Science from the University of Maryland and is a member of the ACM. Mailing address: The MITRE Corporation, 1120 NASA Road 1, Houston, Texas, 77058.

I. INTRODUCTION

Almost all software systems are designed by assembling two teams: the *users*, who know the requirements of the planned system because they are the intended recipients, and the *analysts and developers*, who are skilled in capturing the users' requirements in a form that lends itself to a straightforward implementation. From this point a "generic" or "traditional" methodology in software development begins:

1. The analysts translate the users' problem domain into some sort of abstract data model that serves as a foundation for further analysis.

2. Specific applications against the data model are proposed, approved by the users, and designed.

3. A software prototype may optionally be built to validate the data model and the requirements.

4. The final user interface is proposed, approved by the users, and designed.

5. The system is implemented, delivered, and maintained by the developers.

Two aspects of this traditional methodology deserve special attention. First, the construction of an explicit data model to drive all subsequent development has only recently been emphasized so strongly. This is due in large part to the emergence of a rich theory in entity-relationship (ER) modeling [1] that today forms the core of almost all data modeling methodologies. Second, recent advances in computer-aided software engineering (CASE) technology have greatly enhanced the analyst's productivity by providing an integrated environment from the very beginning stages of conceptual design all the way through to the final stages of system implementation.

Drawbacks of the Traditional Method

Although data modeling and CASE technology have made a substantial impact upon the art of systems analysis, the traditional methodology described above suffers from certain drawbacks that often make it unattractive in its pure form. Five of the most common drawbacks are discussed in this section.

First, data models are complex structures that cannot be designed effectively by inexperienced persons. In fact, it takes a fair amount of training before an analyst can use data models with any degree of competent fluency, and a large part of the training involves repeated exposure to examples and heuristics. Users themselves, then, cannot be expected to learn data modeling skills before they can state their requirements, even though they may be able to understand portions of a proposed data model.

Second, users are notorious for forgetting critical requirements until the user interface has been designed and they can visually confirm that key data items have been omitted or distorted. The overhead of going back to amend the data model may become prohibitive, especially since the unamended model has been serving as the platform for all subsequent development.

Third, the relationship between the users and the developers often has a confrontational flavor. Users generally have zero ability to perceive their own problem domain from a learner's perspective, and as a result the developers are rarely able to digest the users' requirements at a rate which the users find convenient.

Fourth, developers are tempted to view CASE technology as an independent source of creativity rather than as a documentation tool. CASE technology, like all computer technology, has enormous potential to increase human productivity, but it can also automate errors with equal efficiency.

Fifth, and perhaps most important, traditional methods collapse in the face of volatile user requirements. The well-known maxim *Developing software for a given set of requirements is like walking on water: it's easier when it's frozen* illustrates this crisis in current software development paradigms. There is no question that software development is significantly easier when the requirements are well-defined, well-understood, and static. Nevertheless, many users these days are coerced into signing off on specifications documents that they don't understand or, worse, don't even like. Static requirements should be viewed merely as a desirable condition for software development, not as the ultimate goal.

Hence the traditional methods seem to work best in a utopian world where each user knows exactly what he wants, never changes his mind, and can communicate his needs succinctly and patiently to the analysts. Even under such ideal conditions, however, traditional methods would fail for certain broad classes of software systems. These systems are termed *intractable* and are discussed in the next section.

Intractable Systems

More than any other field, perhaps, systems analysis suffers from a shocking disparity between the real world and the world painted by theoreticians and textbooks. The novice analyst is often confronted by mutually uncooperative users who greatly multiply the complexities of requirements analysis. Certain systems may be inherently intractable because of a highly factional user group.

For example, there may be no consensus among the users as to what the exact requirements are. There is perhaps nothing as frustrating for the analyst as spending weeks or months mastering the intricacies of the users' world and then discovering that the users are split into irreconcilable schools of thought on the best way to do things. More often than not, the analysts are helplessly sucked into controversies and then end up alienating themselves from the entire user group.

This pattern has been formally investigated. Several years ago, for example, Curtis, Krasner, and Iscoe, tasked by The Microelectronics and Computer Technology Corporation (MCC), studied the problems of designing large software systems by interviewing personnel from seventeen large projects. Their conclusions contain this highly relevant observation:

> Fluctuation and conflict among requirements afflicted large system development projects continuously... There was a natural tension between getting requirements right and getting them stable. Although this tradeoff appeared to be a management decision, it was just as often adjudicated by system engineers. Fluctuation and conflict among requirements were exacerbated when several organizational components presented themselves as the customer and the developers had to negotiate a settlement [2, p. 1283].

Even when the users are agreed on the general requirements, they may differ on the specifics of the proposed system. There may, for example, be no consensus as to whether the system should automate and improve existing procedures or merely automate them in their unimproved state. Moreover, many elegant systems have been designed, implemented, and then reimplemented because some specific aspect (e.g., security) has not been properly addressed.

These problems greatly amplify the drawbacks of the traditional methodology described above and can easily stall progress until the users can negotiate a unified consensus among themselves.

The proposed technique, object-oriented iterative development (OOID), can often be used to define requirements for these so-called "intractable" systems. OOID minimizes the usual confrontation between analysts and users, and, unlike traditional methods, it exploits the power of object-based prototypes to draw opposing users together and thus to make hidden requirements visible as early as possible. Before OOID can be presented, however, certain concepts relating to iterative development and prototyping must be made more precise.

Iterative Development

For many years the creation of new software systems was performed as a one-pass and one-chance effort. It was fashionable to "initialize" the project by locking all future development into a predetermined and unswerving course. The author recalls his undergraduate days in computer science in the early 1980's, for example, when students were startled by the allegedly sound practice of writing user manuals for systems that weren't even coded yet. There seemed to a "shipbuilding" mentality to software development: detailed systems were created in dry dock under the most rigorous of specifications, and then launched to sink or float in a hostile user environment. If the software "ship" sank, then it was because the requirements hadn't been nailed down by the analysts, or, worse, because the users hadn't communicated their needs. In any case, it seemed almost more important to obey the stated requirements than to discover and fulfill the users' genuine needs.

This was naturally due to unattractive software change economics. The uncompromising rigidity of developers was motivated by the enormous overhead of changing the system once development had started, but that overhead has begun to shrink nowadays with the advent of certain software development paradigms. New "spiral" methods of development have been proposed [3] which recognize the fact that small upstream investments can yield huge

downstream payoffs. There is a growing consensus that complex software requirements are best gathered iteratively by trial and error, and a few voices are even heard to claim that it is impossible to "get it right" the first time [4].

Interestingly, the growth of iterative development techniques seems to have sprung from a small yet omnipresent subculture and not from a sudden widespread recognition of money-saving potential. Rosson, Maass, and Kellogg investigated the use of iteration in 22 software projects (17 from IBM and the remainder from other organizations). Their findings [5] indicate that iterative methods are used surprisingly often, especially when (a) the system is a research project and not a business product, (b) the system is created by small development teams, and (c) the system is developed in a fluid software environnment (e.g., an interpreted programming language). They began their research with the belief that "iteration in design, combined with early and continued user testing, is critical to the design of usable interactive systems." Their conclusions tended to confirm this belief, although they discovered that the concept of iteration is more complex than one might like to think. It should be stressed that the Rosson, Maass, and Kellogg study focussed on interactive systems for unsophisticated users. The benefits of iterative design methodologies for noninteractive systems remain less clear, although intuitively one would suspect that such systems would have more stable requirements. In any case, it is clear that user requirements *for certain systems* can be captured more accurately when design and implementation occur concurrently.

Evidence for this thesis appears time and again in a variety of forms. Perhaps the best example is the notion of the *software release.* Even the most loyal fans of one-pass development strategies are forced to admit that all useful software systems, especially commercial systems, evolve in fixed increments known as releases. New releases are sometimes necessary to correct bugs or to enhance the generality and portability of the system. However, the dominant motive for preparing new releases is to extend or refine the functionality of the system based on a more precise apprehension of user requirements.

Software Prototypes

As previously discussed, prototypes can be fitted into the general framework of the traditional method. Unfortunately, they usually become visible far too late in the development process, and thus their "output" is merely a *yes* or *no* on the question of whether the requirements have been successfully captured.

Much prejudice exists today on the topic of prototypes. Too many software engineers have reported success with prototypes and then are lured into the false belief that prototypes are universally critical to the success of any software project. On the other hand, opponents of prototypes sometimes tend to forget that a software prototype is fundamentally different from other types (e.g., an automobile prototype). This topic will be discussed later in more detail.

Software prototypes, like all other things, have certain attributes. Three of the most important are proposed below and are useful for classification purposes.

First, prototypes may be classified by *purpose.* Note that most prototypes have multiple purposes, so these categories are not mutually exclusive. Some common purposes are:

- *Validation* - To determine whether the user's requirements have been understood and stated correctly. Such prototypes usually revolve around the user interface and procedures by which the final system will be operated. In contrast to purpose (2), this is an "evaluative" use of prototypes.

- *Feedback* - To prod the user into rethinking his requirements and reformulating them if necessary. As this paper later describes, feedback prototypes are an integral part of OOID and can be used with great success on intractable systems. In contrast to purpose (1), this is a "generative" use of prototypes.

- *Feasibility* - To determine whether the user's problem *can* be solved by software in the target hardware environment. This need not be limited to complex or highly technical systems; a feasibility prototype for a simple text editor, for instance, may be necessary to prove that it can function on a given hardware platform within certain response-time tolerances.

- *Propriety* - To determine whether the user's problem *should* be solved with a software-based tool. Many users, in their zeal to computerize everything in sight, request complete software solutions to automate trivial or rarely-used processes. Many of today's useless software systems could have been halted early in development if a suitable propriety prototype had been built, yet the traditional method does not seem to accommodate these kinds of prototypes. This is especially regrettable since many good systems are made inappropriate because of some easily corrected flaw (e.g., a cumbersome user interface).

The currency of many synonyms for *prototype* (e.g., *proof-of-concept*, *pathfinder*, etc) may indicate some sensitivity to these different purposes.

Second, prototypes may be classified by *scope*. In other words, how much of the user's problem do we wish to solve to achieve the purpose of the prototype? There are two general classes of scope:

- *Vertical* - The prototype is a complete, top-to-bottom solution for some very limited subset of the user's problem. Software vendors, for example, routinely release vertical prototypes, often called "demo versions," of their products that provide total functionality for a highly restricted subset of the problem domain (e.g., a spreadsheet that only works on integers in the range of 0 to 10).

- *Horizontal* - The prototype has almost no functionality, but it simulates the final user interface so thoroughly that a naive user may be deceived into thinking he has the real thing. Horizontal prototypes are therefore shallow systems with a very large surface area.

Third, prototypes may be classified by *destination*, or degree of reusability. There are three basic types:

- *Throwaway* - The prototype is built as quickly and as cheaply as possible and is then discarded when its purpose has been fulfilled (zero reusability).

- *Recyclable* - The prototype is built with the intent that parts of it may be recycled or even cannibalized like used-car parts (partial reusability).

- *Embryonic* - The prototype is built with the intent that it will evolve into the target system (total reusability). This classification scheme implies potential reusability, not obligatory reusability. Portions of an embryonic prototype, for example, may be discarded or heavily altered, provided that the prototype is regarded as one stage in the incremental evolution of the system.

Problems with Software Prototypes

Most software engineers seem to understand instinctively the pitfalls of software prototyping. These problems may now be discussed more precisely in the context of the classification scheme given in the previous section. The central issue here is that many prototypes have no charter: they have not been formally classified before they are developed. Unexpected shifts in purpose, scope, and destination during prototyping can bring chaos to the whole software development process.

A shift in purpose sometimes occurs when the users and the analysts have not been communicating properly. It is very common to create a feasibility prototype that turns into a validation prototype, and vice versa. For example, a feasibility prototype may be created merely to show that the the users' problem has a software solution, but the users will view the prototype as evidence that their own requirements have not yet been captured correctly. Generally these types of problems can be overcome by better communication and a more explicit statement of goals.

A shift in scope, or any sort of ambiguities in intended scope, can destroy the user's ability to distinguish the prototype from the final system. For example, a vertical prototype may be initiated and then sidetracked by excessive attention to details of the user interface. Similarly, users may look at a good horizontal prototype and then decide to "see what happens" when more real-world functionality is added. Prototyping efforts are almost certainly doomed to drag on indefinitely unless the scope is explicitly made part of the official charter.

A shift in the destination of the prototype is by far the most pernicious of the three. Many software engineers and programmers are invited to keep writing quick throwaway prototype code, and then one day they suddenly discover that the users wish to copy the software in mass quantities and release it as an operational system. This unfortunate tendency corrupts the central value of prototyping and should either be stopped or harnassed constructively. As described later, OOID attempts the latter by discouraging throwaway code. In any case, this tendency illustrates the fundamental difference between software and almost everything else: software can be duplicated perfectly in any desired volume at nearly zero cost.

Conclusions

The great dilemma of quality software development is as follows: A change in user requirements cannot be managed efficiently, and yet changes must be tolerated if user requirements are to be captured precisely. The key to

solving this dilemma rests in the management of change and not in detailed methodologies for pressuring the users to state their exact requirements once and for all before the start of system implementation.

There is an emerging software technology that, among many other things, addresses the management of change. It is known as *object-orientation* and it is an attractive ally to iterative prototyping because it introduces *change insulation* among the software components of a system.

II. AN OBJECT-ORIENTED PERSPECTIVE

A discussion of basic object-oriented concepts is far beyond the scope of this paper. General introductions can be found in [6] and [7], but one of the best ways to learn object-oriented concepts is to select an object-oriented language and begin using it to solve problems. This transition to a new way of thinking may be smoothed somewhat by selecting an object-oriented version of a familiar programming language (e.g., C or Pascal).

Key Issues of Object-Orientation

However, two key issues of object-orientation are especially germane to OOID and thus deserve some attention in this paper. These issues, *encapsulation* and *inheritance*, empower object-oriented systems to accommodate change far more readily than procedural systems do. The concept of "sturdy changeable software" may seem oxymoronic to some, but within a true object-oriented context such software is not only possible but actually inevitable.

Encapsulation *compartmentalizes* change by hiding implementation details from the outside user. Data and processes, ordinarily regarded as complementary but distinct, are merged into a single atomic unit known as an *object* that presumably acts as a software surrogate for some real-world entity. This is a dramatic generalization of the black-box principle because the user can only tell the object what to do and cannot treat it as a simple input/output module. [This perhaps explains the peculiar expression *Object X knows how to...* that would seem odd in the context of procedure-based programming languages.] Even the data items themselves may be concealed from the outside world. The user may tell an object to remember a new value for a certain attribute (encapsulated data item), or to disclose the current value of an attribute, but the attribute generally may not be accessed as if it were a simple program variable.

Inheritance *facilitates* change by providing easy mechanisms for extension. A new object type may be created simply by declaring the ways in which it differs from an existing object type. All other aspects of the two object types are tightly shared, and any changes to the ancestor type will automatically and instantly change the descendant type. In other words, changes in an object-oriented system are highly localized *unless* distinct object types have been deliberately defined with overlapping parts. In this case changes propagate instantly without explicit intervention to all objects that share those parts.

A Taxonomy of Object-Oriented Concepts

The object-oriented universe of discourse is vast. So many "object-oriented" systems have been propounded in recent years that a brief taxonomy of major issues and concepts is necessary to avoid confusion in terminology. Again, these concepts are presented merely to provide a context for OOID. For more detailed treatment the reader is referred to specific references.

- *Object-Oriented Languages.* Programming languages may be designed to support objects (i.e., the syntax of the language itself provides mechanisms to define objects, encapsulated data and procedures, and inheritance). Following Wegner's terminology [8], a language which simply supports objects (e.g., packages in Ada [9]) is called an *object-based language.* An object-based language that supports class inheritance is called an *object-oriented language* (e.g., Smalltalk [10]).

- *Object-Oriented Databases.* When working in an object-oriented language, the programmer may readily create any number of objects in volatile storage. Unfortunately, these objects generally die when the program terminates unless explicit steps have been taken (e.g., saving the entire workspace to disk). Recalling that *database* may be regarded as any software system in which data elements outlive the programs that created them, we may extend this notion of persistence to the object-oriented world. When an object-oriented software platform supports persistent objects and provides the means to access and update them efficiently, it is called an *object-oriented database system.*

- *Object-Oriented Applications.* Programs may be developed to perform the three database operations implied in the preceding paragraph (storage, query, and update); these programs may be termed *object-oriented applications.*

- *Object-Oriented Systems.* A system may be defined as *any* integrated collection of specialized entities. This definition is quite broad and thus, technically speaking, includes everything discussed in this section. However, the term *object-oriented system* generally refers to any integrated collection of object-oriented databases and object-oriented applications.

- *Object-Oriented Design.* So heavily are object-oriented principles grounded in the theory of data abstraction that Meyer [11] defines *object-oriented design* as "the construction of software systems as structured collections of abstract data type implementations." This differs greatly from the traditional "functional" or "process-driven" design techniques. A functional decomposition of a system identifies what the system does in terms of specific processes, and it relies on models and notations that are totally unlike the data structures of the final implemented system. Object-oriented design, on the other hand, regards the objects of the system as the basis for design, and thus it models the system in a form that closely resembles the final implementation.

  In a narrow sense, object-oriented design is not always limited to an object-oriented implementation. Object-oriented design can occasionally be used as a basis for laying out the structure of conventional systems; the result is then analogous to implementing a highly structured algorithm in a language that does not explicitly support structured programming.

- *Object-Oriented Analysis.* The first step of most software development methodologies involves the analysis of the problem and the construction of various abstract models. Hence the term *object-oriented analysis* may loosely be regarded as any form of systems analysis that (a) leads to object-oriented design, and (b) decomposes the problem domain into object-like entities. It must be stressed that by now we are quite far from our original core of object-oriented topics. One of the author's favorite books, for example, is entitled *Object-Oriented Systems Analysis* [12], yet this book does not even once mention the concepts of encapsulation, information hiding, object classes, or inheritance.

Two other common object-oriented concepts remain to be discussed, although they may be treated as independent topics without a necessary connection to object-oriented analysis, design, or systems.

- *Oriented-Oriented User Interfaces.* As noted by Cox [6, p. 123], iconic user interfaces are generally associated with object-oriented systems and even vice versa. There are at least two good reasons for this linkage. First, icons and other graphic items serve as perfect illustrations of common object-oriented principles. The high degree of specialization in spite of central commonality provides a spectacular example of inheritance and polymorphism. Second, iconic user interfaces are more "what" then "how." This parallels the object-oriented approach and may explain the object-oriented flavor of most icon-based interfaces. In any event, an *object-oriented user interface* is usually visualized as a collection of icon-filled windows, even though this certainly need not always be the case.

- *Object-Oriented Prototyping.* By now it should be clear that (a) encapsulation provides the means to compartmentalize change, and (b) inheritance provides a rigorous framework of generality within which software may be shared and reused. These properties of object-oriented software lend themselves very well indeed to the task of using objects as a host for embryonic software prototypes. Diederich and Milton have discussed the power of experimental prototyping using Smalltalk:

  > Indeed, it [Smalltalk] is an integral tool for promoting experimental prototyping. More explicitly, in our experiments we often made sweeping changes in the design system's architecture, generally with little reprogramming effort, and usually with the introduction of only simple bugs that were easily identified and fixed [13].

  Thus we return again to the problem of change. Traditional software development paradigms do not cheerfully permit "sweeping" changes in software specifications or implementation: changes have a nasty tendency to ripple outward and perturb things far beyond their own localities. Encapsulation, however, insulates software components from each other and thus immediately dampens the ripples of change. OOID is a generalization of the Diederich and Milton approach in that it is used for nonexperimental prototyping and is not limited to Smalltalk.

Having discussed the use of prototypes, the benefits of iterative design, and the multifaceted nature of the object-oriented paradigm, we are now in a position to discuss OOID in detail. OOID is clearly undesirable for many systems in which requirements are well-defined or easily discovered. However, OOID has been used within the

Mission Operations Directorate at NASA's Johnson Space Center for complex budget systems, and it has proven to be a powerful tool for designing intractable systems.

Object-Oriented Iterative Development

There are many paradigms for software development; OOID differs from most of them in the following ways:

1. OOID makes unusually heavy demands on the users, with minimal intervention from the analysts, to reconcile their differing requirements in the form of a detailed and implementation-independent *user model* of the final user interface. This user model, and not necessarily the users themselves, is interrogated by the analysts as they design the data models.

2. OOID recognizes the role of data modeling, but, unlike most information architectures, makes it subordinate to the user model. Moreover, the single critical feature of OOID is that an object-oriented user model, and not data model, drives all development for intractable systems.

3. OOID partitions the entire system into a *user space* and a *data space*. The user space contains all objects visible to the user and it thus realizes the entire user model. The data space is hidden from the users and realizes all entities and relationships in the data model, as well as the physical database itself.

4. The boundary between the user space and the data space is realized by special *boundary objects* that insulate the user model from the underlying data model. This arrangement allows the interface to be fluidly prototyped, and incrementally developed, in the context of specific database views without regard to the actual structure of the data model itself.

5. As the prototyped user model evolves in response to changing requirements, the evolution is forced top-down through the boundary objects into the data model.

6. OOID is an iterative model that incrementally refines itself into the desired target system. A given event within the OOID framework may theoretically be described in terms of conventional software development phases (e.g., logical design, implementation, maintenance), but there are no sequential restrictions placed on OOID activities at any point.

In summary, any system developed under OOID may be viewed as a single huge object. Its external interface (the user space) is defined by the users directly and the internal structures (the data space and boundary objects) are designed and implemented by developers guided by the external interface. Development is strictly governed by object-oriented principles: all prototyping is considered embryonic and done primarily to provoke user feedback in the form of refinements to the user model.

The User Model as a Basis for Software Development

As described in the previous sections, the user model is a centralized representation of the proposed system's user interface. It is visible to both the users and the developers but may be changed only by the users. It is object-based and is free to evolve as the users' requirements are corrected or refined.

Traditional methods of analysis and design stress the prepotence of the models. Correctly structured data and process models created under these methods are supposed to imply the structure of most desired applications; hence, issues such as the user interface (sometimes called the "presentation architecture") are not addressed until very late in the development lifecycle. OOID takes a diametrically opposite approach. The reason, of course, is that OOID is intended primarily for intractable systems whose data and process models cannot be defined very well without several feedback iterations with the users. The vehicle for feedback is the user model, which supplies the users with a concrete target to manipulate, critique, and refine.

Anything that deviates from the user model does not, by definition, reflect the currently accepted requirements. This policy liberates the analysts from any factional disputes within the user group. Users themselves realize that their goal is to produce a consistent user model for the analysts, and that it is pointless to involve the analysts in dispute resolution because the analysts cannot change the user model. It must be stressed that this policy works only when the user model is expressed in a language or form that all users can easily understand.

How then can a user model be represented? The surprising answer is that it doesn't really matter as long as it defines the behavior of the user interface in concrete terms. The user model may exist informally on paper using ad

hoc notation, or it may be hosted in a large-scale user interface development tool such as [14]. A sample user model is given in the following OOID scenario.

III. TYPICAL OOID SCENARIO

Figure 1 depicts the software architecture of an OOID-based system, and the general OOID process is given as a flowchart on Figure 2. This section will provide a step-by-step description of each OOID event.

The implied problem illustrated by the scenario is as follows: Given the conditions for an intractable software system, use OOID to design and implement the system. The term *system* is used in its conventional sense: a data storage, retrieval, and update facility (database) coupled with software procedures that execute user requirements (applications).

Phase 1: Develop User Model

This first of three phases corresponds to the top third of the flowchart shown in Figure 2. The goal of Phase 1 is to devise a tentative user model (and its implementation, the user interface prototype) to guide all subsequent development.

The first step is to provide the users with all tools and formats necessary to define their user model of the system. As previously described, the user model may be stored and managed in any number of forms, but it must in some way describe the proposed system's user interface and it must be satisfactory to all users. If disputes arise within the user group then they must be resolved within the user group, for the user model is the sole medium for expressing the desired behavior of the system.

The developers will then decompose the tentative user model into a hierarchy of disjoint *interface units* (IUs). Each IU is a logical "bundle" of the user interface that performs primitive tasks such as cursor manipulation or icon movement (but nothing associated with data management). The IU concept is quite general and may be applied to a variety of interface styles. For ordinary noniconic interfaces each IU will normally correspond to a single screen or possibly a text window within a screen; for iconic interfaces each IU may correspond to a single window on a desktop.

Finally, the developers map each IU onto one or more objects (in the object-oriented programming sense) and begin prototyping. Users enjoy complete visibility into the IU prototypes, and eventually they can visually confirm that the user model is being implemented correctly.

From time to time the users will ask whether the nascent user interface is behaving as desired. If the answer is no, then the user model must be adjusted and the IUs must be modified. This iterative process continues until the users are generally satisfied with both the user model and the user interface (as embodied by the prototyped IUs).

The output of this phase is a horizontal embryonic prototype, implemented in an object-oriented programming language, that will continue evolving into the final user interface. There are four key features of this phase. First, the analysts and developers may study the user model in detail at a comfortable pace and are not forced to master irrelevant details of the problem domain. Second, the developers understand only the user model and do not honor any other form of communication from the users. Third, users enjoy a high degree of freedom in exploring their own requirements and are much more willing to consider variant solutions and to accommodate other user viewpoints. Fourth, and most important, the contents of the user space (i.e., the user model and all prototyped IUs) may be reduced to consistent data and process models without significant user participation.

A simple example should help to clarify these ideas. The task of inventory management is a very popular problem often used to illustrate concepts in data management. Suppose our users desire an automated inventory management system and Phase 1 of OOID is about to begin. Each user will first be asked to envisage the user interface of the proposed system.

One of the users describes his ideal interface as a simple screen shown in Figure 3. The user enters a customer number and an order number (an order number itself is not unique to the system) and the interface will display complete information on the customer, the order, and a detailed list of items ordered. Of course, the complete system will embody many dozens of screens like this, as implied by the top of Figure 1, but this user is only interested in querying one highly restricted view of the inventory database. Unlike other users, he will not be proposing vast families of related screens that may require sophisticated navigation.

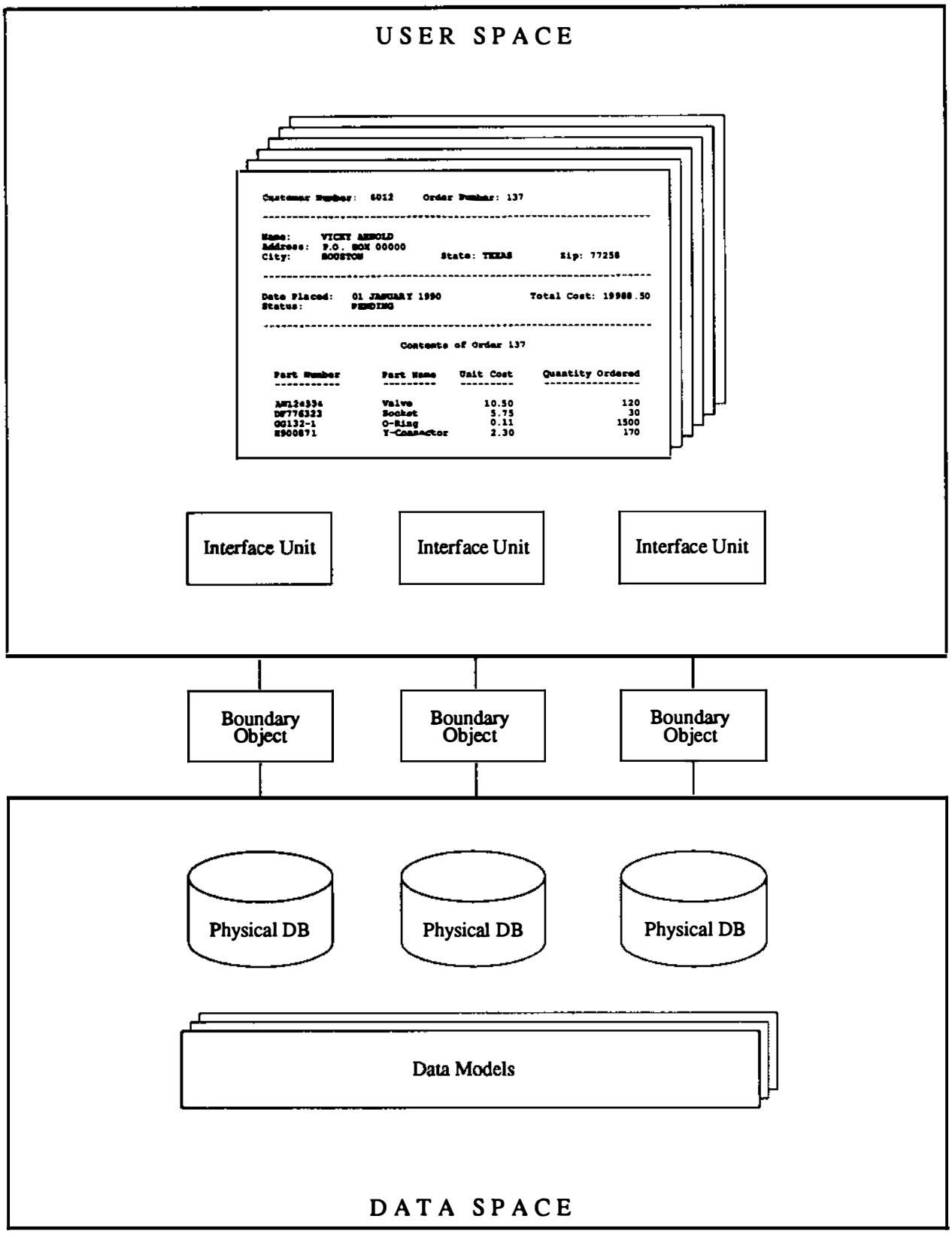

USER SPACE
Customer Number: 6012 Order Number: 137
Name: VICKY ARNOLD
Address: P.O. BOX 00000
City: BOOSTON State: TEXAS Zip: 77258
Date Placed: 01 JANUARY 1990 Total Cost: 19988.50
Status: PENDING
Contents of Order 137
Part Number
Part Name
Unit Cost
Quantity Ordered
AM12d33a Valve 10.50 120
DF776323 Socket 5.75 30
GG132-1 O-Ring 0.11 1500
H900871 Y-Connector 2.30 170
Interface Unit
Interface Unit
Interface Unit
Boundary Object
Boundary Object
Boundary Object
Physical DB
Physical DB
Physical DB
Data Models
DATA SPACE

Figure 1

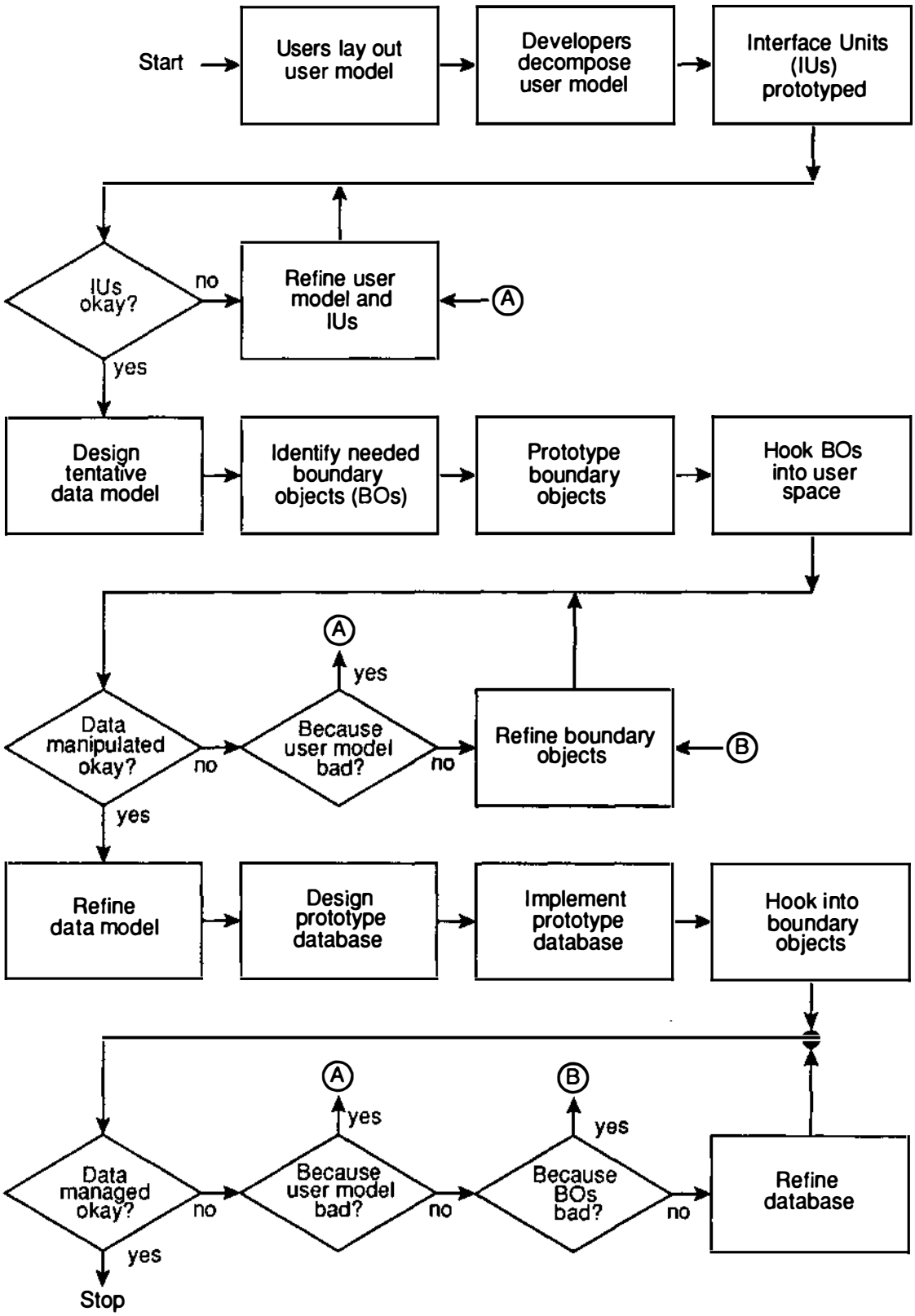

Start
Users lay out user model
Developers decompose user model
Interface Units (IUs) prototyped
IUs okay?
no
yes
Refine user model and IUs
A
Design tentative data model
Identify needed boundary objects (BOs)
Prototype boundary objects
Hook BOs into user space
Data manipulated okay?
no
yes
Because user model bad?
yes
no
Refine boundary objects
B
Refine data model
Design prototype database
Implement prototype database
Hook into boundary objects
Data managed okay?
no
yes
Stop
Because user model bad?
yes
no
Because BOs bad?
yes
no
Refine database

Figure 2

The OOID analyst will consider Figure 3 but will return it to the user as incomplete. Figure 3 may show a specific instantiation of an interface unit, but it does not reveal general properties that must be defined before the IU can be prototyped.

After some reflection, the user will deliver the revised interface shown in Figure 4. Note that the contents of the original screen have now been classified by their role: (a) *constant* data, which are fixed and merely displayed when the screen is refreshed; (b) *user input* data, which the user may change at will; (c) *derived* data, which are retrieved from other regions of the system depending on the current settings of the user input data; and (d) *computed* data, which are calculated at runtime from other data on the IU. It is again stressed that this fourfold classification scheme is *not* an explicit part of OOID; it is simply one of many possible bases for devising a satisfactory user model.

This is now sufficiently detailed to permit the first iteration of prototyping. The OOID developer may design an object **S0001** as follows (using the syntax of an object-oriented Pascal):

```
S0001 = object
        CustomerNumber: integer;
        OrderNumber: integer;
        Procedure Setup;
        Procedure ReadData;
        Procedure SetData (Customer, Order: integer);
        Procedure Display;
        end;
```

Thus any application within the inventory management system may declare an object S of type **S0001** and use it to display the specific contents of a given order. A call to **S.Setup** could clear the monitor display, draw borders, and display all constant data (e.g., titles, labels, and headings). A call to **S.ReadData** will prompt the user to enter a customer number and order number. Alternatively, if the application already knows this information, then a call, say, to **S.SetData(6012,137)** will tell S not to prompt the user for anything but to behave as if customer #6012 and order #137 had been entered. Finally, the application may issue a call to **S.Display**, which displays all remaining data items on the screen and manages the user's commands to scroll up or down, or to exit from the screen altogether.

Of course, the method **Display** of **S0001** (the object class to which S belongs) cannot show any meaningful data yet because that is not the responsibility of the IUs. The very first versions of **S001.Display** may simply throw up random characters. As the system evolves, however, **S001.Display** will presumably begin to call upon the boundary objects for help. This brings us to Phase 2.

Phase 2: Develop Boundary Objects

This phase corresponds to the middle third of the flowchart shown in Figure 1. The goal of Phase 2 is (a) to devise the implied data and process models implied by the user model, and (b) to define and prototype all boundary objects needed to support the user interface.

First, the developers will translate the user model into a tentative data model. This may be done using simple entity-relationship modeling techniques or it may be performed by a full-blown CASE tool. The vehicle for expressing the models is not critical, provided that it is derived exclusively from the user model and not from any additional data from users themselves. If data inconsistencies or irregularities surface during this step, they are referred back to the users, who in turn must amend the user model.

The developers will then turn their attention to the design of boundary objects. A boundary object may be visualized as a concrete realization of some specific view of the data. It is a self-contained, logically coherent bundle of programming effort (i.e., an object) that can move data on demand between the IUs and the physical database. The purpose of the boundary objects is to prevent any portion of the user interface from fusing with low-level database access methods.

Let us return to the illustration with the inventory management system. From the IU in Figure 4 the OOID analyst would probably note the existence of *customer* objects, *order* objects, and *part* objects. For documentation purposes these objects could be modeled as the entities shown in Figure 5 with the <1:m> and <n:m> relationships. If the physical database will be implemented in a traditional database framework (e.g., the relational model), then these data models are usually necessary. On the other hand, the models may be less relevant if the system will be implemented as a true object-oriented database. In either case, it is important to remember that the models are a result, not a driver, of the OOID process.

Customer Number: 6012 Order Number: 137

Name: VICKY ARNOLD
Address: P.O. BOX 00000
City: HOUSTON State: TEXAS Zip: 77258

Date Placed: 01 JANUARY 1990 Total Cost: 1988.50
Status: PENDING

Contents of Order 137

Part Number	Part Name	Unit Cost	Quantity Ordered
AW124534	Valve	10.50	120
DF776323	Socket	5.75	30
GG132-1	O-Ring	0.11	1500
H900871	Y-Connector	2.30	170

Figure 3

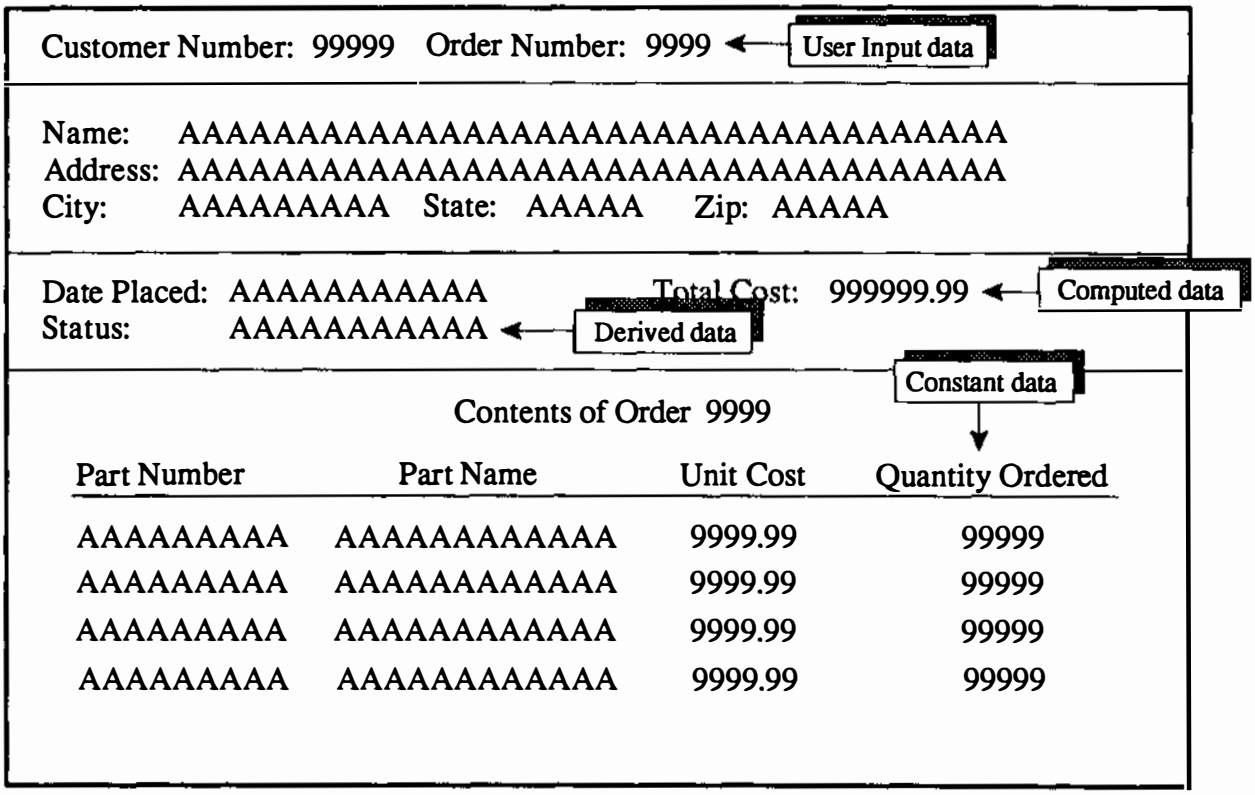


Figure 4

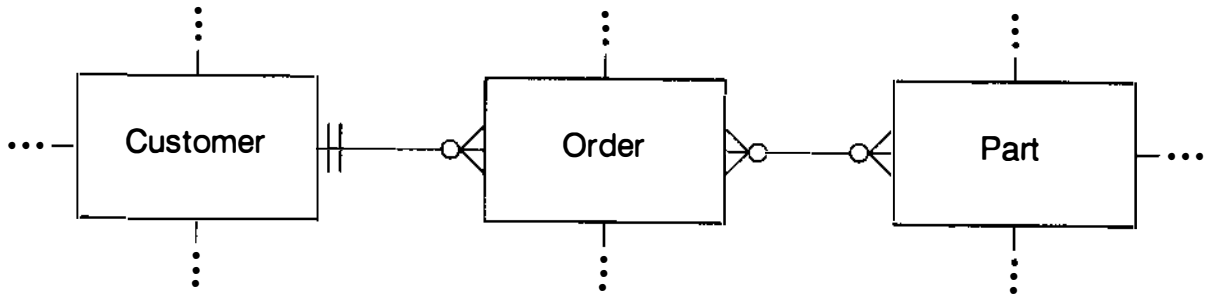


Figure 5

The OOID analyst must first ask what types of boundary objects would be needed to support the given IUs. From the developer's perspective it would perhaps be convenient to assume the existence of a single boundary object:

```
B001 = object
        CustomerName: string;
        CustomerAddress: string;
        CustomerCity: string;
        CustomerState: string;
        CustomerZipCode: string;
        OrderDatePlaced: string;
        OrderStatus: char;
        PartNumber: stringlist;
        PartName: stringlist;
        UnitCost: integerlist;
        QuantityOrdered: stringlist;
        Procedure Retrieve (CustomerNumber, OrderNumber: integer);
        Procedure DisplayCustomer (X,Y: integer);
        Procedure DisplayOrder (X,Y: integer);
        Procedure DisplayContents (Y,Line: integer);
        end;
```

Just as the main applications call the attached procedures of **S001**, so do the methods of **S001** call the attached procedures of **B001**. The method **S001.Display**, for example, could declare a local object **B** of type **B001** and then initialize it by calling **B.Retrieve(self.CustomerNumber, self.OrderNumber)**. The body of **B.Retrieve**, which is not visible, would access the physical database and populate all the attributes of **B001** (e.g., **CustomerName**, **OrderDatePlaced**, etc).

The object **B001** uses two "utility" boundary objects **integerlist** and **stringlist**. These objects presumably store lists of integers and strings, although **B001** is not interested in how these lists are managed. During early stages of prototyping, they may be implemented as fixed-length linear arrays. Later, as the prototype evolves into an operational system, they may be optimized in-place (as a binary tree, perhaps) without perturbing anything higher up. Ideally, these two objects should be reused from a lib ry of useful objects.

The method **B001.DisplayCustomer** simply displays the current customer name, address, etc., at column *x* and row *y* of the screen. How it does this should not be asked: note that **S001.Display** (using a local object **B** of type **B001**) does *not* do something like this:

```
gotoxy(x,y);
writeln(B.CustomerName);
gotoxy(x,y+1);
writeln(B.CustomerAddress);
```

which, of course, would corrupt the change insulation made possible by information hiding. **B001.DisplayOrder** works analogously. In the case of **B001.DisplayContents** the OOID developer felt that there was no need to

parameterize the *x* location of the contents list, so only the *y* location and the first line-item in the scrollable list are necessary.

Hence the body of **S001.Display**, using a local object **B**, would probably contain these statements:

```
B.Retrieve(self.CustomerName,self.OrderName);
B.DisplayCustomer(10,5);
B.DisplayOrder(10,9);
B.DisplayContents(13,1);
```

and then, as the user touched the up and down arrow keys, **B.DisplayContents** would be called again using different values for the argument **line** to effect scrolling.

Note especially that the boundary objects are driven by prototyping convenience and not by formal rules of data modeling. The boundary objects need not be "normalized" as if they were part of a relational data model; this is handled in the data space.

In summary, the prototyped IUs prove that the user interface is *behaving* as the users wish, and the prototyped boundary objects prove that the system is *manipulating data* as the users wish. Phase 2, like the previous phase, may iterate until the IUs and the boundary objects are cooperating properly.

Phase 3: Closure

This phase corresponds to the bottom third of the flowchart shown in Figure 2. The goal of Phase 3 is to implement a physical database to prove that the system is managing real data as desired. Once this is done, the system is said to have reached *closure*.

The first step of Phase 3 is to convert the tentative data models from Phase 2 into a preliminary physical database. A physical database is simply an inert collection of bits coupled with general (usually primitive) access methods. For example, a record-based prototype would include operations to create a record, delete a record, get the next record, etc., for each record type of the system. If the prototype is hosted in an existing database management system, then the developers need only identify the methods that are relevant to the system. If the system is being built from scratch, then these access methods must be programmed explicitly, preferably as objects to preserve change insulation between all objects of the system.

Next, the developers will connect the access methods with all boundary objects. This affects only the encapsulated methods of the boundary objects and thus is totally transparent to the boundary objects themselves and everything higher. Closure is achieved as soon as the boundary objects begin using the access methods to retrieve and update real data from the physical database.

As in previous phases, Phase 3 may iterate until the physical database and the boundary objects are synchronized and performing properly.

Returning one last time to the inventory management system, we see that the system has stabilized enough to begin implementation of the physical database. Reviewing the design of the **B001** boundary objects shows that certain low-level database operations are obviously necessary. **B001.Retrieve**, for example, must be able to fetch all customers, orders, and parts based on the probable keys **CustomerNumber** and **OrderNumber**.

Whether or not this implementation is object-based depends on the nature of the host database management system (DBMS). If the host will be a conventional non-object-oriented DBMS, then obviously there are limits to the influence of object-based techniques. On the other hand, if the physical database is to be implemented in a DBMS that supports object processing, then the connection between the boundary objects and the physical database will be far easier to implement.

What Happens Next?

At this point the prototype is functionally complete and may now accommodate any number of iterations as the system converges to the users' requirements. If the prototype is embryonic, it will continue evolving right into the operational system.

IV. RESULTS OF THE OOID EXPERIENCE

The OOID approach may seem aberrant to some, but each step within OOID is carefully designed to maximize both prototype flexibility and code reusability. As mentioned earlier, OOID has been used to manage software proofs-of-concept at NASA's Johnson Space Center and it has consistently enforced change insulation. This permits the software to evolve very quickly in response to users' demands and yet prevents it from mutating recklessly.

Getting the Users to Work Together

A brief review of the NASA experience may help explain why OOID was so successful. A team of analysts was invited to define requirements for a very complex budget system. A conventional approach was first attempted using elaborate data and process models managed by a well-known CASE tool.

Analysis sessions were conducted regularly and yet failed to produce anything. Outside consultants were brought in to investigate the problem and they suggested a number of actions (defining a more explicit scope, reducing scope, sponsoring more intense modeling sessions with the users, etc) that only seemed to aggravate the problem. Analysts would come close to capturing the problem domain in a comprehensive model, and then a huge part of that model would be voided by a stray remark from one of the users.

Fortunately, the analysis team finally realized that there was no single consistent consensus on what the system should do. Factions within the user group had discovered that the data models were a valid (albeit cumbersome) language for advancing their own organizational goals. This should not be construed as an oblique criticism of the users or of their politics; the NASA budget process is so vast that different perspectives were totally unavoidable.

A breakthrough was achieved when the analysts proposed that the users begin expressing their needs in the form of the desired interface (i.e., a user model) and not in the form of abstract entity-relationship models. This gave the users a sharp focus on their problem, and within days real progress began. The most appealing feature of this change was that the users seemed less and less willing to pull the analysts into their own internal disputes. Previous friction between user factions had occurred quietly and impalpably because the differences were modeled in such abstract forms. Now, however, everybody could see the user interface taking shape before their eyes, and they were able to express their concerns within a much more convenient framework.

There was a second benefit to this sudden emphasis on the user model: the desired behavior of the user interface was known before anything else had been irreversibly modeled. Developers could begin prototyping the interface units and they would get immediate feedback from the users.

It was discovered that minor improvements to the interface were sending shock waves through the data models. For example, perhaps a one-to-one relationship between two entities should actually have been modeled as one-to-many. The "behavior, manipulation, management" prototyping cycle of OOID was seen as a powerful tool for identifying problem areas long before the cost of fixing them became prohibitive. This environment of controlled fluidity drew the users together and greatly reduced the logjams that had dominated earlier analysis sessions.

Advantages and Disadvantages of OOID

OOID is a software development paradigm. Like all such paradigms, it is useful for developing certain types of systems and just as useless for others. The advantages of OOID are summarized as follows:

- Users can express their requirements in a form they understand totally and they are not forced to deal with or approve data models that they may not be comfortable with.
- The analysts and developers are generally freed from the responsibility of arbitrating disputes between user factions. Users can resolve their own disputes if they concentrate on the design of the user model.
- A stable object-oriented environment is provided to support the development of "sturdy changeable software." Objects may be heavily modified or swapped in and out with minimal changes to other objects.
- All software enjoys a very high degree of reusability. The interface units, for example, may continue evolving into the final user interface of the operational system.

- If the OOID prototype is allowed to evolve into the final operational system, then its objects may be transparently optimized without perturbing the overall functionality of the system.

OOID has disadvantages as well:

- Everything is object-based. OOID is difficult to implement except in object-based environments. It may be impossible to prototype the data space, for example, in a conventional database management system that does not support object processing.
- Prototypes in general are clearly not necessary to solve all types of problems. In the software world, some prototypes and proofs-of-concept may be just as unnecessary. OOID adds to development costs whenever requirements are stable and can be easily discovered.
- Users can squabble endlessly over trivial details of the user model. Users on the NASA project, for example, occasionally seemed more concerned with screen colors and upper/lower case spelling conventions than with system functionality.
- Users tend to express their requirements in terms of what they think is available, and it remains the job of the analyst to explore technically elegant alternatives. For example, parochial users tend to design primitive user models, and the analysts may wish to expose such users to more advanced devices (e.g., pop-up menus and dialogue boxes instead of blunt command-line prompts).

Conclusions

Iteration and incremental improvement are essential components in the design of quality systems. Indeed, the premise of this paper is that iteration is the norm for the development of almost everything. Written materials, for instance, routinely pass through iterations, review cycles, and revisions before they are released. This model should hold for software as well, and CASE tools should support it explicitly.

The user model has played a central role in the discussion of OOID, especially in its role as a means to drive out elusive requirements. This technique of getting the users to "think interface" does much to concentrate the mind and can bring structured cooperation to the whole software development effort.

It is obvious that object-oriented programming offers profound opportunities for increased productivity and software quality. It is only natural that object-oriented concepts in general should begin to influence all aspects of software development. OOID represents one step in this direction.

References

[1] Chen, Peter. *The Entity-Relationship Approach to Logical Data Base Design,* Q.E.D. Information Sciences, Wellesley, Massachusetts, 1977.

[2] Curtis, Bill, Herb Krasner, and Neil Iscoe. "A Field Study for the Software Design of Large Systems," *Communications of the ACM,* November 1988, volume 31, number 11.

[3] Boehm, B. W. "A Spiral Model of Software Development and Maintenance," *IEEE Computer,* 21,5 (May 1988).

[4] Gould, J. D., S. J. Boies, S. Levy, J. T. Richards, and J. Schoonard. "The 1984 Olympic Message System: A Case Study in System Design," *Communications of the ACM,* March 1987, volume 30, number 3.

[5] Rosson, Mary Beth, Susanne Maass, and Wendy A. Kellogg. "The Designer as User: Building Requirements for Design Tools from Design Practice," *Communications of the ACM,* November 1988, volume 31, number 11.

[6] Cox, Brad J. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley Publishing Company, 1986.

[7] Thomas, Dave. "What's in an Object?" *Byte,* March 1989, volume 14, number 3, pp. 231-240.

[8] Wegner, Peter. "Learning the Language," *Byte*, March 1989, volume 14, number 3, pp. 245-253.

[9] *Ada Reference Manual*, U.S. Department of Defense, July 1980.

[10] Goldberg, A., and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Reading, Massachusetts: Addison-Wesley, 1983.

[11] Meyer, Bertrand. "Reusability: The Case for Object-Oriented Design," *IEEE Software*, March 1987, volume 4, number 2.

[12] Shlaer, Sally, and Stephen J. Mellor, *Object-Oriented Systems Analysis.* New Jersey: Yourdon Press, 1988.

[13] Diederich, Jim, and Jack Milton. "Experimental Prototyping in Smalltalk," *IEEE Software*, May 1987, volume 4, number 3.

[14] Stott, Jack W., and Jeffrey E. Kottemann. "Anatomy of a Compact User Interface Development Tool," *Communications of the ACM*, January 1988, volume 31, number 1.

Ensuring Software Quality via Formalized Software Processes

Dr. Debra J. Richardson

Information and Computer Science
University of California at Irvine
Irvine, CA 92717
internet: djr@ics.uci.edu
uucp: ...ucbvax!ucivax!djr

Abstract

Quality software products can be developed through systematic software processes that explicitly specify requirements on product quality. Such processes mandate effective testing and analysis of the evolving product to complement synthesis activities. Powerful tools to support such testing and analysis seem essential to assure effectiveness.

The notion of process programming suggests that software development techniques be used to specify not only the objects and tools that are to be integrated into an environment, but also the procedural steps that are to be used to create the objects, the fashion in which objects are to integrated into products, and the assignment of these steps to computing devices and humans. Software tools should be conceived of and developed as devices for creating and altering the software objects that go into the making of overall software products. Software processes themselves should be thought of as software objects that need to be developed through systematic requirements specification, design, coding, evaluation, and evolution processes.

In particular, testing and analysis activities should be developed as software processes in this way. Test planning should be thought of as software development. Test plans should be developed as objects intended to satisfy testing and analysis requirements drawn from software product requirements. In fact, test planning is very much like a product design activity, but should be taken beyond the design phase and should be specified as software process code.

To assure that quality software products are produced, the test planning process should be executed in parallel with the process of developing the software product itself. Quality considerations seem to dictate that the precise way in which these activities parallel each other must also be specified. This is not possible without the use of a formalism such as a process programming language. Quality in software products requires thinking of testing as a software process. Thus, testing and analysis tools should be developed as operators of processes and designed to deal with standardized software objects as their operands.

We have been experimenting with the use of process programming as a mechanism for integrating testing techniques (RAO89). Integration of multiple testing techniques is required to demonstrate high quality of software. Technique integration has four basic goals: reduced development costs, incremental testing capabilities, extensive fault detection, and cost-effective application.

We began this project by specifying testing requirements of adequate coverage and comprehensive fault detection. To accomplish this, we selected two families of testing criteria (Data flow [CPRZ89] and Relay [RT88]) and developed the Bowtie ( ) process program that supports a testing method consisting of any combination of these criteria. Following the process programming philosophy, process development was approached in the same way as software development, i.e., by specifying process requirements, designing the process, and then coding the process in a process programming language, with incremental evaluation and evolution. We found process programming to be effective for explicitly integrating the techniques and achieving the desired synergism. We successfully isolated generic testing tool components and reusable objects [CRZ88]. Developed in this way, testing process programs mitigate many of the problems that plague testing in the software development process.

We are continuing this effort by investigating how to develop testing processes to satisfy specified testing requirements. We are developing means of specifying testing requirements, which include testing criteria, oracle information, and more general quality requirements. We are exploring mechanisms for designing processes that satisfy selected criteria and exploit the synergism among them.

We are also developing processes to assist the tester in carrying out complete testing activities, from automatically selecting test data, driving and monitoring test executions, and comparing results to a test oracle, to triggering regression testing automatically after software descriptions are modified. In conjunction with this work, we are investigating what information must be associated with a test case to enable it to be reused. This reuse of test cases facilitates incremental testing -- that is, testing based on the results of previous testing activities. We are developing a mechanism for maintaining test cases and associated information as well as relating test cases to their selection criteria so they can be effectively reused.

We are also looking at the issue of usability of testing tools and processes. We are developing the Producer system [RAYO90] to prototype analysis and testing processes and evaluate the user "look and feel" of interacting with a process-centered approach to testing. Integrating testing and analysis techniques requires a tester to interact with a variety of testing tools. We are exploring how a tester can interact with these tools in a consistent manner and how the tester can determine the tools to use to achieve their testing requirements.

This talk will describe process programming and the environment support required as well as testing process programs being developed and how these processes address software quality.

Reference

[RAO89] Debra J. Richardson, Stephanie Leif Aha, and Leon J. Osterweil. Integrating Testing Techniques Through Process Programming. In Proceedings of the Third Workshop on Software Testing, Analysis, and Verification, pages 219-228, Key West, Florida, December 1989. ACM Sigsoft.

[CPRZ89] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. "A Formal Evaluation of Data Flow Path Selection Criteria". IEEE Transactions on Software Engineering, SE-15(11), November 1989.

[RT88] Debra J. Richardson and Margaret C. Thompson. The Relay Model of Error Detection and Its Application, In Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, Banff, July 1988. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[CRZ88] Lori A. Clarke, Debra J. Richardson, and Steven J. Zeil. Team: A support environment for testing, evaluation, and analysis. In Proceedings of ACM SIG-SOFT '88: Third Symposium on Software Development Environments, pages 153-162, November 1988. Appeared as Sigplan Notices 24(2) and Software Engineering Notes 13(5).

[RAYO90] Debra J. Richardson, Stephanie Leif Aha, Harry E. Yessayan, and Leon J. Osterweil. Prototyping a Process-Centered Environment. Technical Report TR-90-28, Information and Computer Science, University of California, Irvine, April 1990.

A Formalization of a Design Process

James Kirby, Jr., Robert Chi Tau Lai, David M. Weiss
Software Productivity Consortium
SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070

Software design is frequently viewed as an irrational process, wherein a design can be rationalized after it has been completed [8]. Nonetheless, it is useful to try to formalize the description of the process so that one may understand it better, so that one may provide automated support for it, and so that one may provide guidance to software designers. This paper describes a formal model of a variation of the design process described in [8].

Our model of the design process is a state model; it permits progress in design to be characterized as transitions among states. The model has two levels. The lower level is based on the states of the artifacts produced during the design process; we call such states artifact states *(A-states). In the upper level, states are defined in terms of artifact states, and are augmented with descriptions of activities and the roles of people who may perform activities. Activities include sequences of operations and analyses that can be performed on artifacts within the state. We call the augmented states in the upper level* process states *(P-states). This paper describes the state model, the questions we expect to be able to answer using it, and how we are using it as a guide for the automated support we are developing.*

Keywords: software design process, formal model, state model, software engineering

James Kirby, Jr. is a Member, Technical Staff on the Synthesis Project at the Software Productivity Consortium. Prior to the Consortium he spent three years on the Faculty Technical Staff at The Wang Institute of Graduate Studies and a number of years trying to practice software engineering. He received a Masters of Software Engineering from The Wang Institute. He is interested in software engineering in general, in software design in particular, and in how to foster innovation. Mr. Kirby is a member of the IEEE Computer Society and the ACM.

Robert Chi Tau Lai received the B.A. degree in Architecture in 1981 from Chung-Yuan University, Taiwan, and the M.S. in Architecture in 1985 from the Design Information Processing Laboratory, the Carnegie-Mellon University. He joined the technical staff of the Software Productivity Consortium in 1988, where he is a member of the methodology project. He has worked on object management, Ada task taxonomies, and domain models for software design. His current assignment is to help define the methodology and explore supporting technology for the Consortium's synthesis approach to software development. His principal research interests are design automation methods and tools for software development and architecture design.

Prior to joining the Software Productivity Consortium, Mr. Lai was a visiting research scientist in the school of computer science at Carnegie-Mellon University where he did research on large scale spatial knowledge representation and symbolic computations for geometry and topology. He was a research assistant and manager of the Design Information Processing Laboratory at the Department of Architecture. He also did research on the cognitive process involved in design and on natural language interfaces for computer graphics systems. Mr. Lai is a member of the IEEE Computer Society, ACM, and AAAI.

David M. Weiss received the B.S. degree in Mathematics in 1964 from Union College, and the M.S. in Computer Science in 1974 and the Ph.D. in Computer Science in 1981 from the University of Maryland. He joined the technical staff of the Software Productivity Consortium in 1987, where he has been the leader of the methodology and measurement project. This project has had the responsibility for defining the methodology to be supported by the Consortium's tools. His current assignment is to define the methodology underlying the Consortium's synthesis approach to software development. His principal research interests are in the area of software engineering, particularly in software development methodologies, software design, and software measurement.

Prior to joining the Software Productivity Consortium, Dr. Weiss spent a year at the Office of Technology Assessment, where he was co-author of a technology assessment of the Strategic Defense Initiative. During the 1985-1986 academic year he was a visiting scholar at The Wang Institute. During and prior to his appointment at the Wang Institute, he was a researcher at the Computer Science and Systems Branch of the Naval Research Laboratory (NRL). At NRL he was head of a section that conducted research in software engineering, a member of the Software Cost Reduction project, a consultant to various Navy software development projects, and a participant in a variety of software engineering research projects. He has also worked as a programmer and as a mathematician. Dr. Weiss is a member of the IEEE Computer Society and of the ACM.

1. Introduction

Software design is frequently viewed as an irrational process. Frequent changes in decisions and the large volume of information that must be digested to understand a design prevent the step-by-step, orderly derivation of a design from requirements [8]. One result is that an orderly presentation of the rationale for design decisions is rarely produced. Information that might be used to reconstruct such a derivation is either lost during the process or recorded haphazardly. Recapture of such information after the design is completed is either expensive because of search costs or impossible because the information has been lost. It is only when design is a systematic, repeatable process that there is a chance to systematically record critical information during the process.

Systematizing the design process includes identifying the information needed both during the process and after the process has been completed, and specifying where information of different types must be recorded. Accordingly, it is useful to try to formalize the description of the design process so that one may understand it better, so that one may provide automated support for it, and so that one may provide guidance to software designers concerning what to do at each step and where to record critical information. In this paper we describe an approach to modeling design processes, and give an example of the application of the approach by presenting a formal description of a part of a particular design process. We believe that the same approach can be applied to many design processes.

The Software Productivity Consortium is developing a software design process and tools to support that process. The process we are developing is a variation of the process described in [8]. This paper describes our formal model in order to give an example of the application of formal modeling techniques to the design process.

There are several different approaches that have been proposed for modeling software development. For our purposes we will classify them into procedural models and state models. Procedural models, e.g., [6], assume that the design process can be modeled as parallel sets of sequential events, similar to a set of cooperating sequential processes [3], with each process represented by a program. A design activity, such as producing an interface specification, is defined by a program, which can be executed to simulate the activity. State models, e.g., [5], assume that the process can be characterized as a set of state machines that are executing in parallel, with each state corresponding to a set of activities to be performed by the designer(s). An activity such as producing an interface specification is then one of a number of different states of the process.

Because of the complication in the underlying process, and the amount of backtracking and invention involved, we do not believe that any formal model will both completely and accurately describe software design. Our goal is to be accurate in describing what the work products of design are, what the designers can and cannot be working on, and what the criteria are for completing the work products. We will be incomplete in that we will not attempt to prescribe a total ordering on the events that compose the process. Rather, at points in the process, we will specify that any one of a set of events may next occur. Because state models lend themselves to describing a partial ordering of events better than procedural models, we use a state model. In particular, a state model allows one to define events that permit transitions to predecessor states to represent cases where the designer must go back and redo some or all of a design.

2. The Two Level State Model

We consider a process to be a sequence of decision making activities. Software design is a process of making decisions about (a) what programs should implement the software requirements and (b) what the required properties of those programs are, including properties such as their structure and their interfaces. Information about the decisions made during the design process is captured in artifacts. Example artifacts are a description of the decomposition of a design into a set of components (such as modules, objects, packages, or subroutines) and a specification of the interface of one of the components. To characterize the state of a design we must characterize the state of the artifacts produced during the design process, e.g., whether or not the design decomposition is complete. Just characterizing the state of the artifacts is insufficient to describe a complete design process, however. We must also describe the activities that may be performed on artifacts, the conditions under which those activities are performed, and the roles of the people who may perform them. For example, activities that might be performed on an interface specification are creating it, checking it for completeness and consistency, and conducting a formal review of it. The review will be performed after the specification has been created and analyzed for completeness and consistency. The reviewers may include designers other than the creator, implementors, and quality assurance personnel.

Software design processes vary in many ways, including the way the software is organized into parts, the relations used to define dependencies among the parts, and many other factors. A particular design process may be defined

as a sequence of decisions made in different states. A design methodology may be modeled by a set of predefined states, i.e., it is a prescription for the artifacts to be used, the activities to be performed and their sequencing, and the roles that people play. The definitions of a set of states are the interpretation of the methodology. A designer using the methodology will proceed by following the activities prescribed by the states to produce the prescribed artifacts in the prescribed order. Figure 1 illustrates the elements from which we compose a design model; it is derived from [9]. Arrows in the figure indicate a relation between the elements that they connect, e.g., a role is composed of activities, an activity has subactivities, and an artifact has subartifacts.

The work products of any design process, typically documents, are design artifacts. Most design artifacts are composites or aggregates, i.e., they are composed of a number of parts, each of which is also a design artifact. At some level of organization, the parts may be considered elementary, i.e., for the purposes of design, they are not further decomposable. As an example, Figure 2 is an excerpt from an interface specification for an information hiding module. It is composed of sections, such as a table of operations, each of which is itself an artifact. An entry in the table describing an operation is an elementary artifact.

Our goal in identifying artifacts as part of a design method is to find artifacts whose state can be easily assessed. Furthermore, we would like to be able to associate a rationale with an artifact. Whereas the artifact represents decisions that have been made, we think of the *rationale* for the decision -- what issues led to the making of the decision, what alternatives were considered, and what was the justification for the alternative chosen — as being a separate concern from the decision recorded by the artifact. The rationale for the decision should be captured elsewhere [10].

Our state model has two levels. The lower level is based on the states of the artifacts produced during the design process; we call such states *artifact states* (A-states). Because A-states alone are insufficient to describe completely the design process, we augment them with descriptions of activities, operations on artifacts, and analyses that can be performed on artifacts within the state, as well as roles of people involved. We call the augmented states in the upper level model *process states* (P-states). A complete description of the state model is presented in a later section.

The two level model allows us to separate the description of the process from the representation used for the artifacts. For example, the upper level model may specify operations to be performed on interfaces to information hiding modules [7] without specifying the representation of such interfaces. (Typical operations to be performed might include creating the specification and baselining the specification.) The specification of the representation may be confined to the lower level model, where the state of the artifact that represents such a module is defined. Figure 3 shows the relationships between P-states, A-states, and the other elements of the design model. The bolded entities and relationships in the figure indicate what has been added to Figure 1. The relationships referenced in Figure 3 will be defined in the following section.

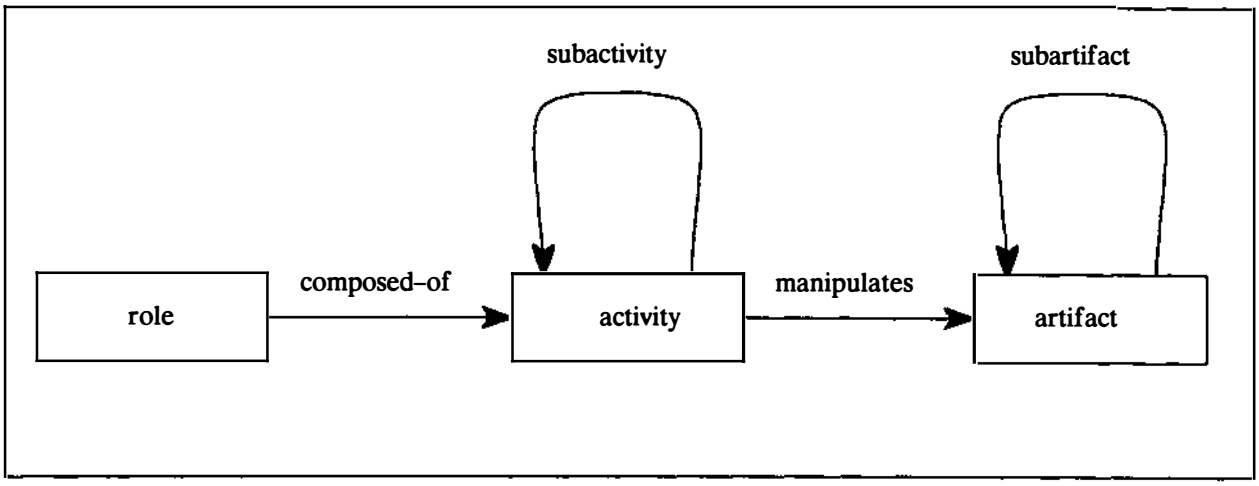


Figure 1
Elements of A Design Model

Sufficiency Requirements: Users of this module need to:

1.. add line. Add a new *line* to the *line list*.

2.. del line. Delete a specific *line* from the *line list*.

3.. chg line. Change the contents of a specific *line* in the *line list*.

4.. ret line. Retrieve the contents of a specific *line* in the *line list*.

5.. trav lines. Traverse the elements in the *line list*.

Feasibility Requirements

1.. store line. There is a mechanism for internally storing and manipulating lines of text.

Operations			
Name	Parameter	Parameter Info.	Undesired Events
LINES	p1:lineno:O	*# lines*	
WORDS	p1:lineno:I p2:wordno:O	*line* *# words* in *line*	LINE_LIMIT_EXCEEDED
CHAR	p1:lineno:I p2:wordno:I p3:charno:O p4:char:I	*line* *word* *character*	LINE_LIMIT_EXCEEDED WORD_LIMIT_EXCEEDED CHAR_LIMIT_EXCEEDED
CHARS	p1:lineno:I p2:wordno:I p3:charno:O	*line* *word*	LINE_LIMIT_EXCEEDED WORD_LIMIT_EXCEEDED
SETCHAR	p1:lineno:I p2:wordno:I p3:charno:O p4:char:I	*line* *word* *character*	LINE_LIMIT_EXCEEDED WORD_LIMIT_EXCEEDED CHAR_LIMIT_EXCEEDED
DELINE	p1:lineno:I	*line*	LINE_LIMIT_EXCEEDED
DELWRD	p1:lineno:I p2:wordno:O	*line*	LINE_LIMIT_EXCEEDED

Effects	
Operation	Description of Effect
SETCHAR	if lineno > 'LINES' then LINES := 'LINES' + 1 if wordno > 'WORDS(lineno)' then WORDS(lineno) := 'WORDS(lineno)' + 1 CHARS(lineno,wordno) := 'CHARS(lineno,wordno)' + 1 CHAR(lineno,wordno,charno) := char
DELINE	LINES := 'LINES' −1 For all r greater than or equal to lineno For all wordno For all charno WORDS(r) := 'WORDS(r+1)' CHARS(r,wordno) := 'CHARS(r+1,wordno)' CHAR(r,wordno,charno) := 'CHAR(r+1,wordno,charno)'
DELWRD	WORDS(lineno) := 'WORDS(lineno)' − 1 For all r greater than or equal to wordno For all charno CHARS(lineno,r) := 'CHARS(lineno,r+1)' CHAR(lineno,r,charno) := 'CHAR(lineno,r+1,charno)'

Figure 2 Excerpt from Line Storage Module Interface Specification

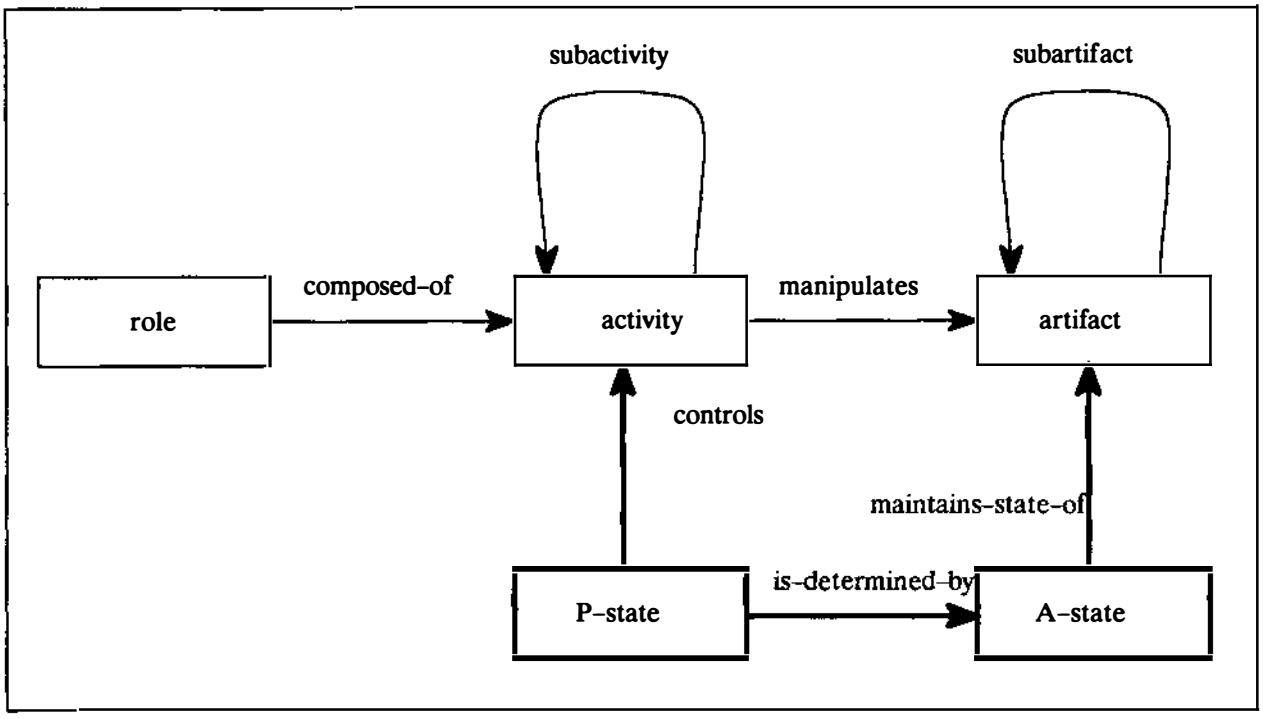


Figure 3
Relationships Defining the Design Model

For different methods, the design artifacts used are different. For example, in our design method, specifications of information hiding modules ([1], [2]) play a key part. We consider an activity to be the work that goes into producing or manipulating a design artifact, i.e., each activity is associated with a particular artifact. (Activities whose design artifacts are composite have subactivities, each associated with a component artifact.) Accordingly, our model is specialized to our design artifacts.

Whether an activity is performable or not depends upon the state of the design artifacts. At any point in time, the set of performable activities represents the choice of artifacts on which the designer may work. Those that are not performable represent the artifacts on which he may not work. Our model prescribes a permissive ordering on activities (and on the work of the designer) by specifying which activities are performable and which are not at any point. By permissive we mean that the designer is free to choose from among the set of performable activities those to pursue. We specify a permissive ordering because we want to support a realistic process. The model captures the information needed to produce a rational description of the work products of an opportunistic process [4].

In addition to specifying artifacts and activities, we also specify the roles played by people involved in design. As with artifacts and activities, different design processes will specify different roles, such as designer, reviewer, implementor, and manager. The roles are defined by the activities in which they may participate.

By augmenting state descriptions with activities and other process-related information, we are able to analyze the needs of the designer at any point in the process. The goal of the analysis is to allow us to present to the designer the information that he needs at the time that he needs it. It also helps us focus on the concerns of what guidance to give to the designer and what analyses can be performed on the design at any time. Finally, it helps us separate the concern of what information should be presented to the designer from how the information should be presented to him.

3. The Artifact State Model

In this section we describe the A-state model for the Consortium design method. We begin by describing the design artifacts. Then we define the A-states that we use to specify the states of design artifacts. Finally, we provide examples of applying A-states to an elementary artifact (i.e., one that contains no other artifacts) and to a composite artifact (i.e., one that is composed of other artifacts).

3.1. The Design Artifacts

The design artifacts for the Consortium design method are listed in Figure 4. Indentation and decimal numbering are used to indicate which artifacts are aggregations of other artifacts. From the figure we see that the design is an aggregation of four types of artifacts: information hiding structure, dependency structure, process structure, and abstract interface. The first three types of design artifacts are descriptions of design structures. The fourth is a specification of the interface to a component of the design, based on the abstract interface described in [1], [2], and [8]. Figure 2 is an excerpt from such an interface.

The information hiding structure describes decisions about what can and cannot be easily changed. It describes the decomposition of our software into components called *modules*. The dependency structure describes decisions about what artifacts are affected by changes to other artifacts. This information is recorded by the *depends on* relation. If for design artifacts *A* and *B*, (*A*, *B*) is in the relation then we say that *A* depends on *B*; *A* can be present and correct if and only if *B* is also present and correct. *A*, then, is affected by changes to *B*. A part of the dependency structure describes which design requirements depend on other design requirements. An excerpt of the Keyword In Context Index (KWIC) design requirements dependency structure is illustrated in Figure 5 (our examples are taken from a design for the KWIC described in [7]). An arrow from one design requirement to another indicates that the former depends on the latter. The process structure describes the decomposition of the executing software into a set of sequential programs.

The information hiding structure is an aggregation of modules and the is-composed-of relation. Each module is an aggregation of a name and a changeable decision. One can think of the changeable decision as defining the responsibility of the module: the module is responsible for encapsulating the changeable decision. In the KWIC example, the changeable decision of the Line Storage Module is how to store, maintain, and access a list of lines. The module hides the data structures and algorithms used to store, maintain, and access them. The abstract interface for the module provides programs (see Figure 2) that other programs may use to manipulate a list.

The is-composed-of relation, a relation among the modules, defines a tree. A module in the tree is composed of its children. The modules at the leaves of the tree are work assignments for individual programmers or for small teams of programmers. The assignment is to design and implement the module. To make the assignment precise and to identify clearly the decision encapsulated by the module, the programmer produces an interface specification for the module. The module's implementation must satisfy the specification. The interface specified is an abstraction of the changeable decision. For that reason, the interface is known as an abstract interface. Line Storage (see Figure 2) offers to its users an abstraction of a list of lines via its abstract interface.

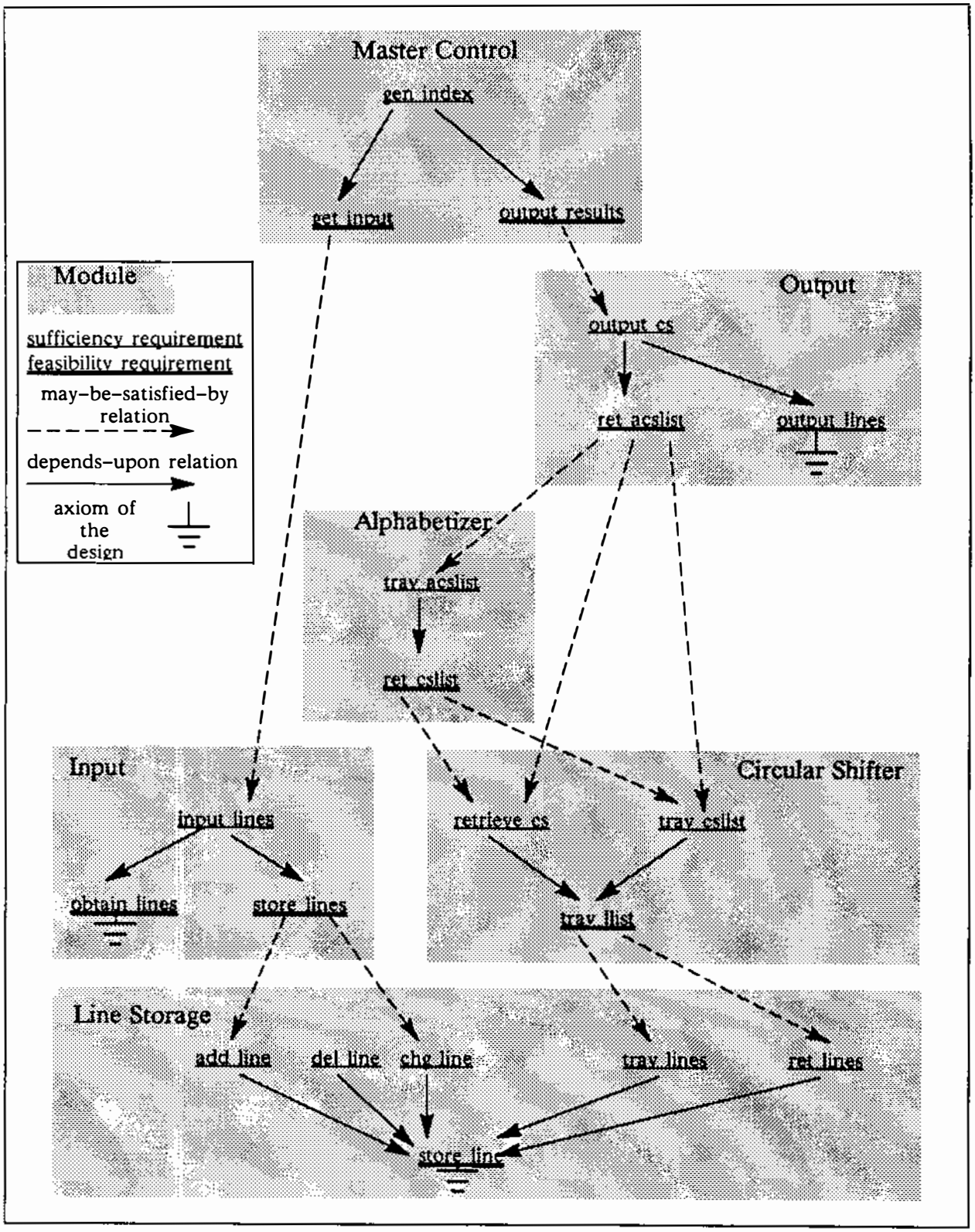
Recall that design artifacts record decisions made by the designer. The abstract interface records decisions about what is visible to users of the module, what users can depend upon, and what might change. Figure 6 lists the artifacts that are components of the abstract interface and the design decisions that each records. Figure 7 provides examples of some of these artifacts taken from Line Storage. As shown in Figure 4, some of these components of abstract interface are also aggregations of other artifacts. For example, in Figure 7 the operation has subartifacts that are specifications and descriptions of two parameters (*ρ1* and *ρ2*) and a reference to an undesired event (*LINE_LIMIT_EXCEEDED*).

We augment the model presented in Figure 3 with design rationale [10]. The augmented model is shown in Figure 8. What has been added to Figure 3 is bolded. The relations in Figure 8 are defined in Figure 9. As part of the design and review activities, issues about artifacts are created and commented on. An issue raises a question about the appropriateness of some aspect of an artifact. For example, a reviewer of the dependency structure in Figure 5 may raise an issue about sufficiency requirement del line, "This sufficiency requirement is not used, can we delete it?" Others might respond to the issue with *yes* and *no* positions, giving arguments, respectively, that "Eliminating del line will make the design simpler." and "Keeping del line will make the design more reusable." Prior to the completion of the design, the designer selects a position for each issue and, if necessary, changes the design to reflect the chosen position. Once an issue is created it is considered to be *open* until the designer selects a position and brings the design into conformance with the position.

Design

1. Information Hiding Structure
    1.1. Module
        1.1.1. Name
        1.1.2. Changeable Decision
    1.2. Is-Composed-of Relation
2. Dependency Structure
    2.1. Design Requirements Dependency Structure
        2.1.1. Is-Implemented-by Structure
        2.1.2. Depends-Upon Structure
        2.1.3. Operation-Depends-Upon Structure
        2.1.4. May-be-Satisfied-by Structure
    2.2. Uses Structure
3. Process Structure
4. Abstract Interface
    4.1. Name
    4.2. Sufficiency Requirement
    4.3. Feasibility Requirement
    4.4. Operation
        4.4.1. Name
        4.4.2. Parameter
            4.4.2.1. Name
            4.4.2.2. Type Reference
            4.4.2.3. Mode
            4.4.2.4. Description
        4.4.3. Undesired Event Reference
    4.5. Effect
    4.6. Is-Implemented-by Matrix
    4.7. Depends-Upon Matrix
    4.8. Anticipated Changes
    4.9. Anticipated Subsets
    4.10. Performance Specification
    4.11. Accuracy Specification
    4.12. Type Glossary
        4.12.1. Type Definition
    4.13. Undesired Event Glossary
        4.13.1. Undesired Event Definition
    4.14. Technical Term Glossary
        4.14.1. Technical Term Definition

Design Artifacts
Figure 4



Example Design Requirements Dependency Structure
Figure 5

Figure 6: Decisions Recorded by Components of Abstract Interface	
Artifact	Decisions Recorded
sufficiency requirement	What is required by users of the abstract interface
feasibility requirement	What implementors of the abstract interface may assume
operation	How the design requirements may be satisfied by a set of operations
effect	What is the behavior of the operations
is-implemented-by matrix	Which operations implement which sufficiency requirements
depends-upon matrix	Which sufficiency requirements depend upon which feasibility requirements
anticipated change	How the abstract interface may change
anticipated subset	What subsets of the abstract interface will be useful
performance specification	What is the required time performance of operations on the abstract interface and what are the space requirements of the module
accuracy specification	What is the required accuracy of output values provided by operations on the abstract interface
undesired event glossary	What classes of abnormal behavior the module will detect
type glossary	What types are required by operations on the abstract interface
technical term glossary	What are the definitions of terms used in specifying the abstract interface

Figure 7: Examples of Components of Abstract Interface (from Figure 2)				
Artifact	Example			
sufficiency requirement	add line.	Users of this module need to add a *line* to the *linelist*.		
feasibility requirement	store line.	There is a mechanism for storing, retrieving, and manipulating lines of text.		
operation		Parameter	Description	Undesired Event
	WORDS	p1:lineno:I	*line*	LINE_LIMIT_EXCEEDED
		p2:wordno:O	*# words in line*	
is-implemented-by matrix	(add line, setchar)			
depends-upon matrix	(add line, store line)			
anticipated change	We may want to allow the user to maintain more than one *linelist* concurrently.			
anticipated subset	BASIC = (add line ret line)			
undesired event glossary	LINE_LIMIT_EXCEEDED.	p1 is not a valid lineno.		
type glossary	lineno.	Handle for a *line*.		
technical term glossary	*linelist*.	An ordered set of *lines*.		

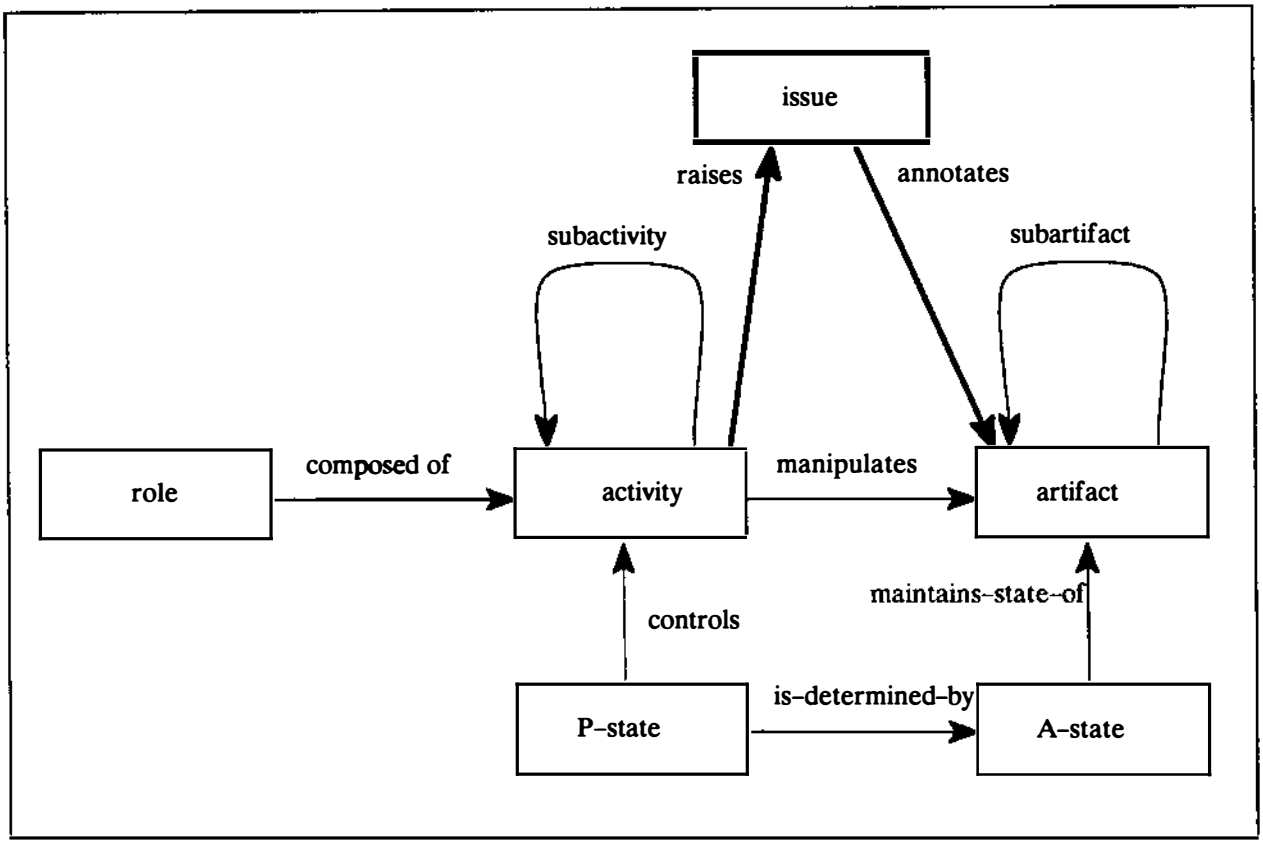


Figure 8
Relationships Defining the Design Model

Figure 9: Definitions of Design Model Relationships			
Relation	Domain	Range	Definition
annotates	issue	artifact	The issue raises a question about the design of the artifact. (many to many)
composed of	role	activity	Someone playing the role may perform the activity. (many to many)
controls	P-state	activity	Whether the activity is performable is determined by the P-state. (many to many)
is determined by	P-state	A-state	Whether the design process is in the P-state is determined by a relation on the A-state. (many to many)
maintains state of	A-state	artifact	The state records information that defines the current state of the artifact.
manipulates	activity	artifact	The artifact is an output of the activity. (one to many)
raises	activity	issue	The issue is an output of the activity. (one to many)
subactivity	activity	activity	The activity in the domain is composed of the activities in the range. (one to many)
subartifact	artifact	artifact	The aggregate artifact in the domain is composed of the artifacts in the range. (one to many)

Figure 10: Design Artifact States	
State	Definition
REFERENCED	There is a reference to the artifact, but there is neither a name nor any work associated with it.
CREATED	There is either a name or work associated with the artifact.
STARTED	There are both a name and work associated with the artifact.
DEFINED	There are both a name and work associated with the artifact and the work either depends upon no other artifacts or any artifacts that it depends upon are DEFINED.
SPECIFIED	There are both a name and a specification for the artifact and the specification either depends upon no other artifacts or any artifacts that it depends upon are SPECIFIED.
DEF&SPEC	There are a name, a specification, and other work associated with the artifact. The specification either depends upon no other artifacts or any artifacts that it depends upon are SPECIFIED. The other work either depends upon no other artifacts or any artifacts that it depends upon are DEFINED.
DONE	There are a name, a specification and/or other work associated with the artifact. If there is a specification it either depends upon no other artifacts or any artifacts that it depends upon are DONE. If there is other work it either depends upon no other artifacts or any artifacts that it depends upon are DONE. The artifact has been reviewed and there are no open issues associated with it.

3.2. The Artifact States

We have defined a set of seven A-states that formalize the state of completeness of each of our design artifacts. The A-states are listed and informally defined in Figure 10. Every one of our design artifacts is in one of these A-states. The A-state of an artifact describes the completeness of the artifact. We define a partial ordering relation on the A-states (illustrated in Figure 11). In the figure, an arrow from *A* to *B* indicates that an artifact in A-state *B* is more complete than an artifact in A-state *A*. We think of the A-states as defining a state machine for each design artifact. Actions performed by the designers and other participants in the design process cause the state machines for the artifacts to transition from one A-state to another. We use the A-states, a formalization of the completeness of the artifacts, to guide us in completing the design. We provide more precise and formal definitions of artifact state when we discuss the individual design artifacts.

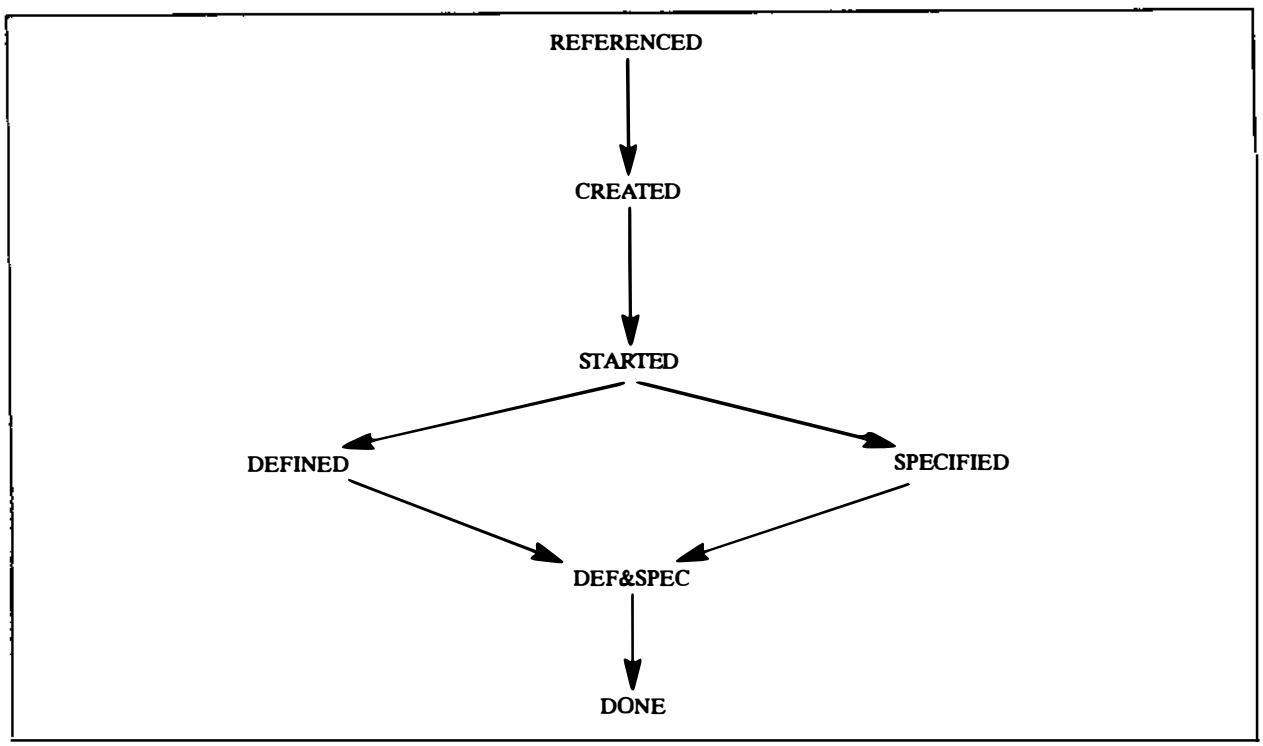
Because we have defined a partial ordering on the A-states, we can use the relational operators *greater than*, *less than*, *greater than or equal*, and *less than or equal* on A-states. For example, for A-states *A* and *B*,

$$B > A$$

means that an artifact in A-state *B* is more complete than an artifact in A-state *A*.

The state machines for some of the artifacts will not include all of the A-states, because those artifacts can never be in some of the states. For example, we write sufficiency requirements in natural language. The sufficiency requirement add_line from the Line Storage Module is "Users of this module need to add a *line* to the *linelist*." Because it does not make sense to say that the sufficiency requirement has been specified, the state machine for a sufficiency requirement does not include the A-states SPECIFIED and DEF&SPEC.

In the preceding discussion of artifact state we have focused on the set of A-states we have chosen for our design process. We would like to distinguish between what is true of A-states in general and what is true of the set of A-states that we have defined to model our design process. For any design process that one wishes to define using our two level state model, one specifies a state machine for each artifact in the process. For a particular process, one may use the seven A-states that we have defined or one may want to define another set of A-states that are better suited to the process. The A-states that make up the state machine for each artifact may be the same for all of the artifacts (as was the case with our process) or they may be different. It is convenient and leads to a simpler model if all of the state machines use the same set of A-states, but for some processes it may not be desirable or possible to use the same set of A-states for every artifact.



Partial Ordering of Artifact States
Figure 11

3.3. The States of Design Artifacts

In the following paragraphs we describe how to determine the state of an artifact. First, we discuss how the state of an elementary design artifact is determined. Then we discuss how to determine the state of an aggregate design artifact.

The state of an elementary design artifact, one that is not an aggregation, is determined by examining the artifact itself, whether there are any open issues that annotate it, and the state of artifacts on which the artifact depends. Figure 12 defines the A-states (and the A-state machine) for a simple artifact, a sufficiency requirement. Predicates used in Figure 12 are defined in Figure 13. A small example of how a sufficiency requirement transitions among the A-states is illustrated in Figure 14.

A sufficiency requirement is a component of an abstract interface. It is a concise statement of what is required by users of the abstract interface. A sufficiency requirement may depend upon other artifacts, which may affect its state. For example, if a sufficiency requirement references a technical term that is not at least in the DEFINED state, the sufficiency requirement can not be in the DEFINED state. Informally, the sufficiency requirement can't be considered to be defined unless all the technical terms it uses are defined. As shown in Figure 12, a sufficiency requirement is in the DEFINED A-state if it has a name, is not empty, and either refers to no other artifacts or any artifacts to which it does refer are DEFINED. A sufficiency requirement is in the DONE state if it satisfies the conditions required to be DEFINED, has been subjected to a technical review, has not been changed since the review, has no open issues associated with it (the output of a review), and either refers to no other artifacts or refers only to artifacts that are DONE.

Figure 14 illustrates how a sufficiency requirement and two technical term definitions upon which it depends transition through A-states. In the figure, underlining indicates a sufficiency requirement name (add line) and italics indicates a technical term name (*linelist* and *line*). Note that when add line is first defined it is in the DEFINED A-state because it refers to no other artifacts. When the definition is modified to include a reference to the undefined *linelist*, add line transitions to the STARTED A-state.

Figure 12: State Condition Table for a Sufficiency Requirement (sr)	
State	Condition
REFERENCED	$(\exists x)(Refer(x, sr) \wedge \sim Name(sr) \wedge Empty(sr)$
CREATED	$Name(sr) \vee \sim Empty(sr)$
STARTED	$Name(sr) \wedge \sim Empty(sr)$
DEFINED	$Name(sr) \wedge \sim Empty(sr) \wedge$ $(\sim(\exists y)Refer(sr, y) \vee (\forall y)(Refer(sr, y) \Rightarrow StateOf(y) >$ STARTED$))$
DONE	$Name(sr) \wedge \sim Empty(sr) \wedge Rvw(sr) \wedge \sim OI(sr) \wedge$ $(\sim(\exists y)Refer(sr, y) \vee (\forall y)(Refer(sr, y) \Rightarrow StateOf(y) =$ DONE $))$

Figure 13: Definitions of Predicates	
Predicate	Definition
Empty(x)	There is no work associated with the object x. The object is empty.
Name(x)	There is a name associated with object x.
OI(x)	There are open issues associated with the object x.
Refer(x, y)	Object x references object y.
Rvw(x)	The object x has not changed since it was subjected to a technical review.

Figure 14: Artifact State Example for Sufficiency Requirement			
Action	State of Artifact Following Action		
	Add Line	*Linelist*	*Line*
1. Sufficiency requirement created and named: add line.	CREATED		
2. Add line defined: "Users of this module need to add to the linelist."	DEFINED		
3. Add line redefined: "Users of this module need to add to the *linelist*."	STARTED	REFERENCED	
4. Technical term definition *linelist* created, named, and defined: "An ordered set of lines."	DEFINED	DEFINED	
5. Add line redefined: "Users of this module need to add a *line* to the *linelist*."	STARTED		REFERENCED
6. *Linelist* redefined: "An ordered set of *lines*."		STARTED	
7. Technical term definition *line* created, named, and defined: "An ordered set of words."	DEFINED	DEFINED	DEFINED
8. Add line, *linelist*, and *line* are reviewed and there are no open issues associated with them.	DONE	DONE	DONE

Figure 15: State Condition Table for Abstract Interface (ai)	
State	Condition
REFERENCED	$(\exists x)(\text{Refer}(x, ai) \wedge \sim(\exists x)(\text{comp}(x, ai) \wedge \text{StateOf}(x) > \text{REFERENCED})$
CREATED	$(\exists x)(\text{comp}(x, ai) \wedge \text{StateOf}(x) > \text{REFERENCED}) \wedge$ $(\exists x)(\text{comp}(x, ai) \wedge \text{StateOf}(x) <= \text{CREATED})$
STARTED	$(\forall x)(\text{comp}(x, ai) \Rightarrow \text{StateOf}(x) >= \text{STARTED}) \wedge$ $(\exists x)(\text{comp}(x, ai) \wedge \text{StateOf}(x) = \text{STARTED})$
DEFINED	$(\forall x)(\text{comp}(x, ai) \Rightarrow \text{StateOf}(x) > \text{STARTED}) \wedge$ $(\exists x)(\text{comp}(x, ai) \wedge \text{StateOf}(x) = \text{DEFINED})$
SPECIFIED	$(\forall x)(\text{comp}(x, ai) \Rightarrow \text{StateOf}(x) >= \text{SPECIFIED}) \wedge$ $(\exists x)(\text{comp}(x, ai) \wedge \text{StateOf}(x) = \text{SPECIFIED})$
DEF&SPEC	$(\forall x)(\text{comp}(x, ai) \Rightarrow \text{StateOf}(x) >= \text{DEF&SPEC}) \wedge$ $(\exists x)(\text{comp}(x, ai) \wedge \text{StateOf}(x) = \text{DEF&SPEC})$
DONE	$(\forall x)(\text{comp}(x, ai) \Rightarrow \text{StateOf}(x) = \text{DONE}) \wedge$ $\text{Rvw}(ai) \wedge \sim \text{OI}(ai)$

Figure 16: Definitions of Predicates	
Predicate	Definition
Comp(x, y)	Artifact x is a component of aggregate artifact y.
OI(x)	There are open issues associated with the object x.
Refer(x, y)	Object x references object y.
Rvw(x)	The object x has not changed since it was subjected to a technical review.

Figure 17: Artifact State Example for Line Storage Abstract Interface				
Action	State of Artifact Following Action			
	Abstract Interface	Add Line	*Linelist*	*Line*
	DONE			
Sufficiency requirement created and named: add line.	CREATED	CREATED		
Add line defined: "Users of this module need to add to the linelist."	DEF&SPEC	DEFINED		
Add line redefined: "Users of this module need to add to the *linelist*."	CREATED	CREATED	REFERENCED	
Technical term definition *linelist* created, named, and defined: "An ordered set of lines."	DEF&SPEC	DEFINED	DEFINED	
Add line redefined: "Users of this module need to add a *line* to the *linelist*."	CREATED	CREATED		REFERENCED
Linelist redefined: "An ordered set of *lines*."			CREATED	
Technical term definition *line* created, named, and defined: "An ordered set of words."	DEF&SPEC	DEFINED	DEFINED	DEFINED
Add line, *linelist*, and *line* are reviewed and there are no open issues associated with them.	DONE	DONE	DONE	DONE

The state of an aggregate design artifact is determined by whether there are any open issues that annotate it, the state of its components, and the state of artifacts upon which the aggregate artifact and its components depend. The state of the aggregate artifact is bound by the minimum of the states (determined by the partial ordering illustrated in Figure 11) of those artifacts that compose it and those artifacts upon which the aggregate artifact and its components depend. For example, the A-state of an aggregate artifact can be no greater than CREATED if a subartifact is REFERENCED. The aggregate artifact can be no greater than DEFINED if a subartifact is DEFINED. We chose the minimum because we want to point the designer toward what might be done next to move the design toward completion.

Figure 15 defines the A-states for the aggregate artifact abstract interface. Predicates used are defined in Figure 16. Figure 17 illustrates how an abstract interface transitions through A-states as the A-states of its components change. In the first row of the table, the abstract interface is in the DONE A-state, because all of its components are DONE, the abstract interface has been reviewed, and there are no open issues associated with it. In the following rows we see how the A-state of the abstract interface changes as a new sufficiency requirement and two new technical terms are added to the abstract interface and changed. In the last row of the table we see that a successful review (*i.e.*, one that results in no open issues) will move the new sufficiency requirement, the two new technical terms, and the abstract interface to the DONE A-state.

3.4. Summary

In this section we have discussed A-states and the A-state machine associated with each design artifact. We have provided examples of A-states, A-state machines, and design artifacts. We have shown how the A-state machines transition as the designer changes the associated artifacts. For our design process, we defined seven A-states that we use to describe the state of each of the artifacts in our process. The seven A-states that we defined are convenient for understanding, managing, and applying the process. We believe that the seven states that we have chosen correspond well to intuitive notions of completeness. The small number of states used throughout the description makes for a simpler, more understandable model. We can formally define the A-states for each artifact which will reduce the ambiguity in trying to assess the state of completeness of a design.

4. The Process State Model

4.1. Process States

The components of a P-state are shown in Figure 18. A P-state is characterized in terms of the A-states of the artifacts associated with the P-state. Consequently, the specification of a P-state includes a list of the artifacts associated with it. Figure 19 is an example P-state, named Abstract_Interface_Design. It describes a state in which work may progress on designing an abstract interface. Accordingly, it is characterized in terms of the artifact *design.abstract_interface*. (The notation for naming the artifact shows where in the design artifacts hierarchy (Figure 4) the artifact belongs.)

In addition to artifacts associated with the P-state, its specification also describes the precondition for state entry, and activities that may be performed in the state. The precondition is the condition that must be true to enter the state. Activities are of two types: operations, which may be used to change the state of an artifact, and analyses, which return information about the design in general, including guidance on design methods, and artifacts in particular.

Figure 18: Components of Process State (P-State)

Components	Definitions
Name	The name of the state.
Precondition	Predicates on A-states that determine when the process enters the P-state.
Artifacts	A list of artifacts on which work may proceed in the P-state
Activities	A list of operations and analyses that may be performed in the P-state. See Figure 20 and Figure 22 for detailed descriptions of operations and analyses.

As an example, in the P-state Abstract_Interface_Design, the operations Create, Edit, Review, and Baseline may be performed on the abstract interface artifact. Any of these may change the A-state of the artifact, e.g., successful conclusion of the Baseline operation may change the A-state of the abstract interface to DONE. (See section 3 for a discussion of the A-states of an abstract interface.) On the other hand, the analyses Consistency, Dependency, Undefined_terms, and Completeness may also be performed on the artifact. These analyses return information about the artifact, e.g., the Completeness analysis will provide information about components of the artifact that are not yet created. The operations and analyses associated with a P-state are described in further detail in Figure 20 and Figure 22, respectively.

Figure 19, Figure 21, and Figure 23 show an example of a P-state. The example is not meant to be complete, but just to illustrate how our process model can be applied.

In specifying predicates in the example, we have made liberal use of functions that return information about the design. For example, the function *state_of* is a state retrieval function that returns the states of design artifacts. For the purposes of this paper, we will assume these functions are primitive, and provide only informal definitions of them as needed in discussing the example. The following sections discuss the components of a P-state in more detail.

4.1.1. Operations

An operation may be specified by a pre_condition, the design artifacts to which the operation may be applied, actions, roles, and post-actions, as shown in Figure 20. Pre_conditions are represented for operations the same as for P-states; post-actions are represented as lists of actions.

An action may be performed when the pre-condition for the operation is true. The action embodies the effect of the operation. Accordingly, the description of an action is a formal specification of the effects of the operation. As in other examples, we have assumed the actions are primitive functions and not defined them here in detail. Since not all design operations are permissible to all the people who participate in design, an operation also has a list of roles of people allowed to perform the operation.

Figure 21 defines the operations for the example P-state Abstract_Interface_Design. The Create operation can only be performed when there is a module with the same name as the abstract interface, and the module is at least in the Created state. Informally, a module must first exist before an abstract interface can be created for it. The Create operation can be performed by a designer, operates only on an abstract interface, and its effect is defined by the function create_abstract_interface. It has no post-actions associated with it.

Figure 19: An Example Process State (P-State)

Components	Values of (Sub)components
Name	Abstract_Interface_Design
Pre_conditions	(state_of(design.module) >= STARTED) ∧ (state_of(design.module) < DONE) ∧ name_of(design.module) = name_of(abstract_interface)
Artifacts	Design.Abstract_Interface
Operations	Create, Edit, Review, Baseline
Analyses	Consistency, Dependency, Undefined_terms, Completeness

Figure 20: Components of an Operation

Components	Definitions
pre-condition	Predicates on A-states that determine when the operation is active.
design artifacts	The design artifact manipulated by the operation.
action	The function to be performed by the operation.
roles	A list of roles of people who are allowed to perform this operation.
post-action	A list of actions to be performed after the operation is completed.

Figure 21: Operations for Abstract_Interface_Design State		
Components	Sub-Components	Values of (Sub)components
create	pre-conditions	((state_of(design.module) >= STARTED) ∧ name_of(design.module) = name_of(abstract_interface) ∧ (state_of(design.abstract_interface) <= REFERENCED)))
	design artifacts	design.module.abstract_interface
	actions	create_abstract_interface(design.abstract_interface)
	roles	designer
	post-actions	none
edit	pre-conditions	(state_of(design.abstract_interface) >= CREATED)
	design artifacts	design.abstract_interface
	actions	edit_abstract_interface(design.abstract_interface)
	roles	designer
	post-actions	none
review	pre-conditions	(state_of(design.abstract_interface) >= STARTED)
	design artifacts	design.module.abstract_interface
	actions	review_abstract_interface(design.abstract_interface)
	roles	designer, reviewer, manager
	post-actions	none
base_line	pre-conditions	(state_of(list_of_dependent_artifacts_of(design.abstract_interface)) >= DEF&SPEC)
	design artifacts	design.abstract_interface, list_of_dependent_artifacts_of(design.abstract_interface)
	actions	baseline (design. abstract_interface)
	roles	manager
	post-actions	none

Figure 22 Components of Analysis	
Components	Definitions
Design artifact	A list of design artifacts related to this analysis.
Analysis	Specification of the analysis to be performed.
Roles	List of roles of people who may perform the analysis.
Context	The context of the analysis
Trigger	How the analysis can be invoked.
Action	Actions to be performed depending upon the results of the analysis.

Figure 23: Example Analyses

Name of Analysis	Sub-Components	Values of (Sub)components
consistency	design artifact	design.abstract_interface

	analysis	function ana_abs_int_const(design.module)
		(for module_var in design
		(if use(module_var, design.module)
		then
		for operation_var in module_var
		if (~consistent(design.module.abstract_interface,
		module_var.operaion_var))
		then collect(result,(design.abstract_interface,
		module_var.operation_var))
		if use(design.module, module_var2)
		then for operation_var2 in design.module
		if (~consistent(design.module.operaion_var2,
		module_var2.abstract_interface,))
		then collect(result,
		(module_var2.abstract_interface,
		design.module.operaion_var2)))
		return result)

	context	syntactic
	trigger	@T(user_requests_consistency_analysis) V @T(state_of(design.abstract_interface) = Defined)
	action type	prompt_message(err_info56), write_to_file(file_name_var), mail_manager(message29,managers_list_var), mail_designer(message_22,designer_list_var), mail_reviewer(message_82, reviewer_list_var)
	roles	designer, manager, reviewer
undefined_terms	design artifact	design.abstract_interface
	analysis	function undefined_terms (design.abstract_interface) (for word_var in design.abstract_interface if high_lighted(word_var) & ~in_design.dictionary(word_var, design.module.dictionary) then collect(word_var, result) return result)
	context	textual
	trigger	@T(user_requests_undefined_terms_analysis) V @T(state_of(design.abstract_interface) = Defined)
	action type	prompt message(message44), write_to_file(file_name_var, format61), mail manager(message77, managers_list9), mail_designer(message33, designer_list_var), mail_reviewer(message78, reviewer_list_var)
	roles	designer, manager, reviewer

4.1.2. Analyses

In a P-state, a designer can get information about the current status of the design by invoking an analysis. The information offered by the analysis helps designers make decisions. An analysis is defined by the design artifacts to which it applies, the functions it performs, its types (context type, trigger type, and action type), and roles of the people who may invoke it, as shown in Figure 22. The type of each analysis is determined by the following attributes:

1. Context:
   a representation of the basis for the analysis, such as quality assurance rules, syntactic analysis, or semantic analysis. For example, a consistency analysis of an abstract interface might be based on the syntax of the interface, i.e., the analysis might verify that all sections of the interface existed and that every technical term used in the interface was defined in a technical terms glossary.
2. Trigger:
   a representation of the events that can trigger the analysis, such as a direct request by the user. An event is a point in time at which a certain condition become true, e.g., entry into a P-state, exit from a P-state, at the expiration of a time interval, or when an artifact enters or exits a particular state. The example in Figure 23 uses the notation used in [2] to specify events. For a predicate *p*, we use @T(*p*) to specify an event. @T(*p*) specifies the point in time when *p* goes from false to true.
3. Action Type:
   Actions may be specified as part of the analysis definition for the cases when certain conditions are found to be true from the analysis. Example actions include (*state* is the state of a design artifact):
   a. prompt message (*state, message*) :
      prompt the designer(s) with the *message* when the analysis returns *state*.
   b. save(*state, name*) :
      save the result of analysis in *name*, when the analysis returns *state*.
   c. mail manager(*state, list of user_ID, message*) :
      mail to the manager(s) (*list of user_ID*) the *message* when the analysis returns *state*.
   d. mail designer(*state, message*) :
      mail to the designer(s) the *message* when the analysis returns *state*.
   e. mail reviewer(*state, list of user_ID, message*) :
      mail to the reviewer(s) (*list of user_ID*) the *message* when the analysis returns *state*.

Figure 23 illustrates a set of analyses for the P-state, Abstract_Interface_Design.

4.1.3. Roles

A role defines the responsibilities of jobs associated with software design, such as designer, manager, or reviewer. A person who plays a designated role can determine what activities he may perform by looking at all the operations and analyses permitted for that role over all states. We expect that roles will vary according to the design method used, e.g., a reviewer's role in one design method may be quite different than in another design method.

4.2. The Process Model As A State Machine

Because designers are likely to be working on several aspects of a design concurrently, we model the design process as a concurrent state machine, i.e., a state machine that can be in several states simultaneously. This permits a designer to make progress in one state for a while and then move to another without completing the first. It also permits several designers to work on different parts of the design concurrently. For example, in our design method, definition of a module precedes specification of its abstract interface. Once several modules have been defined, work may proceed on their abstract interfaces concurrently. We model this situation by creating a P-state for each module and for each abstract interface and by requiring as a pre-condition that a module must be defined before work may proceed on its abstract interface, as specified in the example in Figure 19. Our state machine may then be in several of these states at the same time. By offering this capability, the process model can allow designers freedom to act opportunistically, yet still provide a means for guiding them in an orderly way. In addition, it can be used to record an opportunistic design process so that the design can later be presented in a rational way.

Modeling the design process as a concurrent state machine allows us to satisfy several concerns. First, it gives us a precise definition of our design methodology. Designers, reviewers, and others involved in the process have

a reference that specifies what they can do at any stage of the process. Second, it permits a parallel design process, since the model identifies opportunities for different designers in a team to work concurrently and independently. Third, it gives us a specification for a design tool to support the process, i.e., a tool that could keep track of currently active design states, that could record information about the design, and that could provide the necessary operations and analyses for the designer to make progress on the design.

4.3. Questions That Could Be Answered By Our Formal Model

A primary motivation for constructing our model was to answer the following questions at any stage of the design process for software designers using our methods:

1. What do I do next?
2. What output do I produce?.
3. What input do I need?
4. How do I know when I'm done?
5. Who has responsibility what?

To the extent that the model provides capabilities for performing design activities, such as operations and analyses, it also answers the question,

6. How do I do it?

5. Constructing A Model

The previous sections have discussed the elements of the formal model and shown examples of A-states and P-states. In this section we describe an approach to constructing a model for a particular process. To complete such a model requires that the modeler have a clear idea of the artifacts and activities that are used in the process that he is modeling. While the modeler may not have a sufficiently clear view of the process at the outset, one intent of our modeling process is to help him obtain such a view. The following steps form a bottom up approach to constructing a model that may help clarify the model under construction. As with most processes associated with software development, some backtracking will be needed in following this process.

1. Define a set of design artifacts according to the methodology to be used.
2. Identify dependency and composition relations among the artifacts.
3. Use the A-state model to identify a set of states for each design artifact.
4. Identify a set of operations that can be performed on each design artifact.
5. Identify pre-conditions and post actions for each operation.
6. Identify a set of analyses that software developers may need for making decisions in conjunction with each operation.
7. Identify guidelines and methods for each operation, and where possible, translate them to analyses.
8. Group the operations according to those that need to be done about the same time.
9. Collect the analyses, guidelines, and operations together according to the grouping in step 8. to form a P-state.
10. Identify pre-conditions for each P-state.

The result of this process is a set of A-states and P-states that have a flat structure. For particularly complicated models, it may be useful to introduce a hierarchical structure into the set of P-states. For example, if a complete software development process were being modeled, then design might be represented at the top level of the model as a single P-state. The design P-state might itself then be decomposed into a hierarchy of P-states. For such cases, we add an additional step to the modeling process:

11. Identify super-P-states that are groups of P-states to build a hierarchical machine.

6. Using the Model to Build Design Tools

6.1. Building Design Tools

A two-level model of a design process can be viewed both as a specification for the process and as a specification for tools to be used to support the process. The model specifies both the data on which the tools must operate (i.e., the set of design artifacts and their interdependencies), the operations to be performed by the tools, and the effects of those operations (i.e., the state changes caused by invoking the tools). Accordingly, a particular model may be used to explain the process to software developers who will use it, to tool developers who will build tools to support it, or to tool integrators who will integrate a set of existing tools to support the process.

We have deliberately left the type of specification to be used in describing operations and analyses to the discretion of the modeler. For different purposes and audiences, different types will be appropriate. For the purpose of describing the process to software developers, the modeler may use an informal prose specification or a formal specification that describes effects in terms of state changes on composite artifacts. As an example of the latter, the effect of using an editor to create an abstract interface might be specified as

state_of(abstract_interface) := CREATED

For the purpose of describing the services offered by the editor to the software developer, the modeler may provide a reference to the user's manual for the editor. For the purpose of describing to a tool builder the interface provided by the editor, the modeler may reference an interface specification for the editor. For the purpose of specifying the tool to be built, the modeler may reference a requirements specification for the editor. Any combination of such references and specifications may be used by the modeler, as appropriate.

In particular, the implementor of the tool must implement actions that are implicitly specified by the model. For example, in cases where there are dependencies among design artifacts, a state change in one artifact may cause state changes in other artifacts, as shown in Figure 14. Furthermore, there may be informal parts of the model that are not well-captured by the formal model, e.g., specifying methods for resolving contending requests to update the same design artifacts by several different designers. Such requests may arise directly, e.g., when several designers are simultaneously attempting to manipulate the same design artifact, or indirectly, when several designers simultaneously change the states of dependent design artifacts.

One way to make implicit actions more explicit for the purposes of implementation, or to specify conflict resolutions, is to provide a way of specifying actions to be taken on the occurrence of certain types of events. One useful such class is post-actions accompanying state exits, particularly P-state exits. For example, on leaving the P-state Abstract_Interface_Design, the following post-action might be taken.

if (state_of(abstract_interface) = DONE) = = > notify(reviewers_of(abstract_interface))

6.2. Questions That Could Be Answered By A Tool

A tool developed to support our model would also be able to answer questions such as:

1. What is the difference between version X and version Y of the software?
2. What modifications have been made since a design artifact reached a particular state, or since a particular P-state was last visited?
3. What modules are affected by this or that particular modification or batches of changes or modifications? Who modified them, and for what reason?
4. What remains to be done to complete a particular state?
5. What are the states of various design artifacts?

6.3. Integrating Design Tools

The process model can be used as an integration model to integrate different design tools to support a design process. This can be done in the following steps:

1. For each operation and analysis in each P-state, select a set of commercially-available tools that can be used to perform the operations and analyses,
2. Design the interfaces that permit tools access to design artifacts and their states,
3. Implement actions as invocations of an existing commercially-available tool using the interfaces.

Using this procedure, the model becomes an instrument for designing a common tool environment.

7. The Benefits of Our Method of Modeling the Design Process

We believe that there are several benefits to our method for developing formal models of software design processes. The method can be applied to a wide range of processes and methods. The formal models produced help to understand the process, to manage it, and to apply the process in developing software designs.

The method can be applied to any design method or process that has well-defined design artifacts and well-defined composition and dependency relations among the artifacts. Well-defined artifacts and relations allow one to formalize the artifact states upon which the formalization of the process states and of the process depend. Well-defined artifacts allow one to specify what artifacts record the design. Well-defined artifacts and well-defined relations among them allow one to specify how to determine the extent of completeness of the design. For artifacts whose completeness can't be formally specified, we use technical reviews and tracking of the issues that result from them to specify completeness.

The application of our method will result in process models that are relatively simple and clear, that can describe realistic, opportunistic processes and traditional waterfall processes, and that can provide useful guidance to the designer applying the process. These characteristics make the models useful to those trying to understand, manage, and follow the process. The models tend to be simple because our two-level approach allows us to separate concerns. For example, we can use the model to think about what is the state of the design without being concerned with how to determine the state. We have shown how we model part of an opportunistic design process. We model a waterfall process by developing an ordering of the process states and by specifying in the precondition of each P-state that it can not be entered until previous P-states have been exited. We can provide the designer with guidance on what to do next based upon the state of the design. Such guidance could be usefully provided either automatically online or in appropriately indexed hardcopy.

Acknowledgements

We are grateful for thoughtful reviews performed by Paul Clements and Jim O'Connor. The example abstract interface was provided by Jeff Facemire and Jim O'Connor.

8. References

[1] Britton, K.H., R. A. Parker, and D.L. Parnas. "A Procedure for Designing Abstract Interfaces for Device Interface Modules," Proc. 5ICSE, pp. 195-204, 1981.

[2] Clements, P.C., R.A. Parker, D.L. Parnas, J.E. Shore, and K.H. Britton. A Standard Organization for Specifying Abstract Interfaces, NRL Report 8815, June 14, 1984.

[3] Dijkstra, E. W. "Co-operating Sequential Processes," Programming Languages, ed. F. Genuys, New York: Academic Press, pp. 43-112

[4] Guindon, R. A Framework for Building Software Development Environments: System Design as Ill-structured Problems and as an Opportunistic Process, MCC Technical Report STP-298-88, Septmeber 18, 1988.

[5] Kellner, M. "Representation Formalisms for Software Process Modeling," Proc. 4 Intntl. Software Process Workshop

[6] Osterweil, L. "Software Processes Are Software Too," Proc. 9ICSE, March 1987

[7] Parnas, D.L. "On the Criteria to be Used in Decomposing a System into Modules," Comm. ACM, Vol. 15. No. 12. pp. 1053-1058.

[8] Parnas, D.L. and P.C. Clements. "A Rational Design Process: How and Why to Fake It, "IEEE Trans. on Software Engineering, February, 1986.

[9] Potts, C. "A Generic Model for Representing Design Methods," Proc. 11ICSE, May, 1989

[10] Potts, C., and G. Bruns. "Recording the Reasons for Design Decisions," Proc. 10ICSE, April 1988.

A Survey of Software Metrics and Their Application to Testing

Warren Harrison

Portland State University
Department of Computer Science
P.O. Box 751
Portland, OR 97297

Abstract

Software Metrics have become an important issue for many organizations. This presentation will survey classes of software metrics and software testing. Representative metrics from each class will be evaluated for an industrial software system.

Biographical Sketch

Warren Harrison is Associate Professor of Computer Science at Portland State University, where he teaches and does research involving software metrics and management of the testing process. He has been active in both the theory and practice of software measurement and management of the testing process for over ten years and has authored numerous papers.

A Survey of Software Metrics and Their Application to Testing

Warren Harrison
Portland State University

The Eighth Annual
Pacific Northwest Software Quality Conference

October 31, 1990
Portland, Oregon

What *is* Measurement?

Measurement is the Process of Associating a Symbol With an Object

Examples of Measurement

Personality Types (Type A, B)

Product Quality (good, better, best)

Temperature (degrees)

Length of a Trip (miles)

Possible Relationships Among Objects

Same, Different
(*nominal*)

Less Than, Greater Than, Equal
(*ordinal*)

Four Units More, Six Units Less
(*interval*)

Twice As Many, Half As Much
(*ratio*)

Applying Measurement to Software

Real Properties of Interest

Reliability

Maintainability

Modifiability

Testability

Correctness

Desirable Software Metrics

Program A is *buggy/not buggy*

Program A *has more bugs than* Program B

Program A *has 10 more bugs than* Program B

Program A *has twice as many bugs as* Program B

By the time we know this, it's too late ...

We Usually Use *Product Measures* as Proxies Since They're Easy to Collect - try to *predict* the real metrics of interest from the product metrics

Product Metrics

Product Metrics Are Measures of Certain Characteristics of the Software Product

Classified According to the Artifacts They Measure:

Token Based Metrics

Control Flow Metrics

Many Others Exist

Hybrid Metrics

Data Flow and Use Metrics

Interconnection Metrics

Application Metrics

These are Generally Termed "Complexity Metrics"

Their Use is Based Upon the Assumption that the Probability of a Piece of Code Having an Error is Directly Proportional to the Complexity of the Code

Token Based Metrics

View Token as a Unit of Text Within the Program

Token Based Metrics Summarize Occurrence of Tokens

Issues:

Defining Tokens

Weighting Tokens

Summarizing Results

Lines of Code

View a Line of Code as a *Token*

Questions:

How do you define a "line"?

Are all lines counted equally?

How do you summarize this information?

Software Science

View Operators, Operands as *Tokens*

How do you summarize?

η_1 - *unique* operators
η_2 - *unique* operands
N_1 - *total* operators
N_2 - *total* operands

Synthesize Basic Token Counts Into a Variety of Metrics

Volume = $(N_1 + N_2) \log_2 (\eta_1 + \eta_2)$

Level = Minimum Volume / Actual Volume

Effort = Volume / Level

An Example

```
if x = y then
    if y < z
        z=0
    endif
else
    z=45
endif
```

operators		operands	
if/then/endif	2	x	1
=	3	y	2
<	1	z	3
		0	1
		45	1

$\eta_1 = 3$ $N_1 = 6$ $\eta_2 = 5$ $N_2 = 8$

Volume = 14 $\log_2$ 8 = 42

Control Flow Metrics

Considers the Sequence of Execution Through the Code

Summarize the Complexity of the Flow of Control

Issues:

Representing Flow of Control

Weighting Control Flow

Summarizing Results

Number of Decision Statements is the Most Popular Control Flow Metric

This is Also a Token Based Metric

Questions:

How do you define a "Decision"?

Are all decisions counted equally?

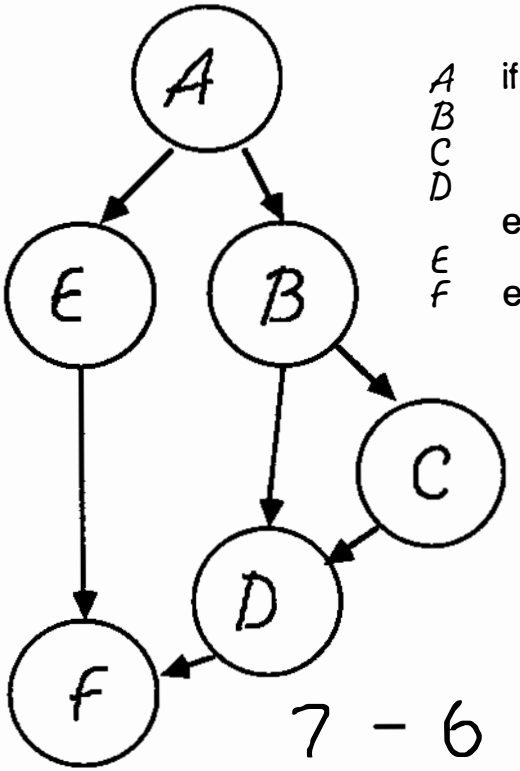
How do you summarize this information?

Cyclomatic Complexity

Number of Basic Paths Through the Program

Flowgraph Calculation: $V(g) = e + n + 2$

Simplified Calculation: number of predicates plus 1



How Well Do the Metrics Work?

Data Collected on 15 C Modules From a Commercial Development Project

27,770 Lines of Code

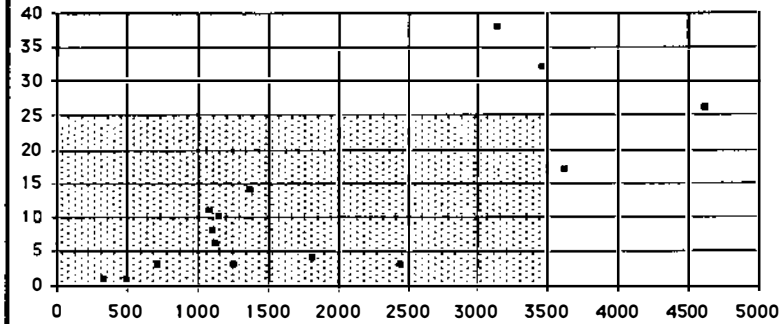
177 Errors Discovered During Testing

Module	Lines	Volume	Vg/Avg	Errors
A	3,459	78,067	208/4	32
B	504	6,609	14/1	1
C	1,158	22,568	89/3	10
D	1,127	23,031	64/2	6
E	3,625	105,029	320/3	17
F	337	2,251	22/11	1
G	2,452	28,616	128/5	3
H	1,816	28,439	115/5	4
I	718	19,205	75/3	3
J	1,371	10,573	33/1	14
K	4,611	92,926	255/3	26
L	3,146	86,635	356/5	38
M	1,112	28,588	82/3	8
N	1,253	17,624	70/4	3
O	1,081	29,695	84/3	11
Average	1,851	38,657	128/4	12
Std Dev	1,294	33,866	107/2	12
Median	1,253	28,439	84/4	8

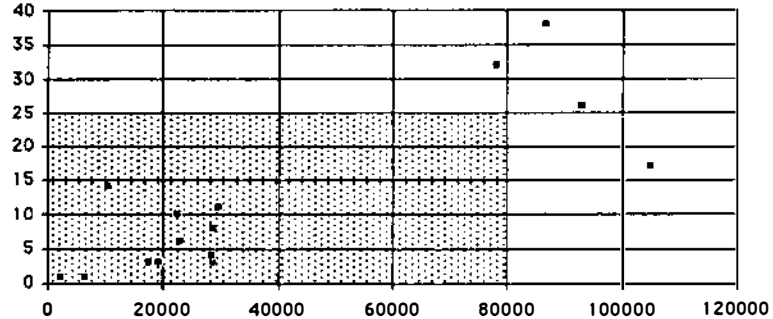
Analyze *Outliers*

Modules Which are More than One Standard Deviation Above the Mean for Complexity or Errors

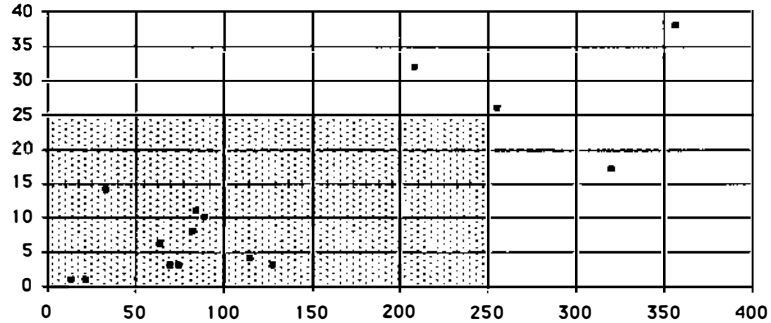
Lines of Code vs. Errors ...



Volume vs. Errors ...



Cyclomatic Complexity vs. Errors ...



Using Metrics in Testing

How Can This Knowledge Be Used???

Threshold Classifications

Identify Modules Which Exceed
the Average Complexity of the Modules in the System
by More Than One Standard Deviation

Do "More Testing" on Modules Which Exceed the Threshold

How Much More Testing???

An Example

LOC Threshold: *3,145* [A,E,K,L]
Volume Threshold: *72,523* [A,E,K,L]
Cyclomatic Complexity Threshold: *235* [E,K,L]

Outliers [A,E,K,L] are Responsible
for 113 out of 177 Errors (64%)
[A,E,K,L] account for 53% of LOC and 62% of Volume

Outliers [E,K,L] are Responsible
for 81 out of 177 Errors (46%)
[E,K,L] account for 48% of Cyclomatic Complexity

Conclusions

Metrics *Can* Help Identify Outliers

Metrics Are Not a Substitute
for Common Sense

There are Many Metrics
Which Metric You Choose
is Not as Important as
Deciding to Choose a Metric

There is a Need for
Tools and Techniques to
Exploit Software Measurement

Reliability Modeling Using Complexity Metrics *

by

Dennis Kafura and Ashok Yerneni

Department of Computer Science
Virginia Tech
Blacksburg, VA 24061-0106
email: kafura@vtopus.cs.vt.edu
phone: 703/231-5568

ABSTRACT

A reliability model incorporating software complexity metrics is presented. The model is studied by simulating a large number of hypothetical systems varying such factors as the mean number of failures per component, the mean complexity, and the correlation coefficient between failures and complexity. The means and variances of the factors in the simulated systems are representative of those found in production software systems. An analysis of the simulation results shows that the model is accurate, stable and directionally correct over the range of factors studied.

Biographical Sketch

Dr. Dennis Kafura has been a faculty member in the Department of Computer Science at Virginia Tech since 1980. His current research interests in software engineering center on the use of software metrics for improving software engineering practices. Other current interests include object-oriented programming, concurrency, distributed systems and real-time systems.

Mr. Ashok Yerneni received his M.S. degree in Computer Science from Virginia Tech in 1989. He received his B.S. degree in Mechanical Engineering from the Indian Institute of Technology in Madras, India. His major areas of continuing interest are user interface design and development, graphics and image processing.

* This research was supported by a grant from the Software Productivity Consortium.

1. Introduction

Software reliability models are useful to both software developers and to end-users. For developers, the model is an essential basis for *assessing* the reliability of a product, *predicting* the change in reliability which would result from a given period or volume of additional testing, and *estimating* the additional resources necessary to achieve a desired level of reliability. For end-users, the model is the, often unseen, basis on which reliability measures are predicated. A user's requirement that a software system exhibit a specified level of reliability invokes implicitly some model of reliability.

Most models for estimating software system reliability are based on the execution behavior observed during system testing. In these models the system being evaluated is treated as a "black box" because knowledge about the system's structure is not used in assessing the system's reliability.

The black-box approach suffers from three limitations:

- available knowledge of the system's design and implementation is not exploited even though they are exactly known and intimately related to reliability;
- reliability is measured only after the system had been completely developed leaving little scope for any major design or implementation changes to improve system reliability;
- no guidance is provided on how to conduct additional testing so that reliability gains are maximized.

Despite these limitations very sophisticated black-box models have been developed (e..g, [Musa 1987]) and used successfully [Ehrlich 1990].

A notable exception to the "black box" reliability models is the model developed by Shooman [Shooman 1976] which uses the frequency of "path" executions, the running time for each path and the probability of error along each path. The times for successful and unsuccessful traversal of each path over N test cases are derived, leading to measures of system probability of failure. The term "path" was used in a general sense to mean any module or function.

The purpose of this study is to explore a software reliability model employing software metrics based on the system's design or implementation complexity and run-time metrics which reflect the degree of testing. The model assumes that each "component" in the system has a measured "complexity." The term "component" means any consistently defined software element which has an unambiguous complexity measure and whose distinct executions can be counted. Examples of components are procedures, objects, modules. The model developed here does not assume any specific complexity metric. Any of the numerous complexity measures defined in the literature (e.g., [McClure 1978], [Henry 1981], [McCabe 1976]) would appear to be suitable Finally, unlike most models in the literature, which use calender time or execution time, our model's "time" domain is the number of test cases. No assumptions are made regarding the execution time of each test case.

To evaluate the model we simulated a number of system descriptions with means and variances representative of those in actual software systems. Specifically, we used data compiled from three Fortran coded systems developed for NASA Goddard [NASA 1982]. The systems, each on the order of 100,000 lines-of-code, were designed and implemented by professional programmers working for the Computer Science Corporation (CSC). The application domain, ground support software, placed heavy emphasis on mathematical calculations associated with orbital mechanics, data manipulation and user interaction. The development was undertaken in conjunction with the Software Engineering Laboratory which, in addition to NASA/Goddard and CSC, included researchers from the University of Maryland. The development activities and work products were extensively measured. The means and variances used in this paper were derived from an analysis of this measurement database.

Complexity metrics are often discussed in relation to "errors" while reliability models are usually discussed in relation to "failures." For the purposes of this paper the term failure is defined as an observed difference between the system's execution behavior and the system's requirements. Failures result from the execution of code which

contains errors. An error is defined as one or more related lines of code in a component which do not correctly implement the component's intended function. Testing is the process of systematically executing the components in a system under varying conditions so as to reveal the presence of errors by observing the occurrence of failures. While the terms failures and errors denote distinct entities they are related and, for simplicity, we will use these terms interchangeably in this paper.

The remainder of this paper is organized as follows. Section 2 presents the reliability model in detail. The simulator is described in Section 3. Section 4 contains an evaluation of the model based on a large number of simulations varying the factors which control the simulation. Finally, conclusions are presented in Section 5.

2. The Reliability Model

The reliability model is presented in four steps: (1) the assumptions made in the development of the model are listed and justified, (2) the initial probability of failures for each component based on its complexity is defined, (3) the changes in probability with testing are given, and (4) the system, as opposed to component, probability of failure is derived.

The principal assumptions on which the model is based are summarized in Table 1 and discussed below. Several other assumptions are given in [Yerneni 1989] and will not be discussed here.

Table 1: Model Assumptions

1	The probability of discovering an error in a component is inversely proportional to the degree of testing to which the component has been subjected without a failure.
2	The probability of discovering an error in an untested component is directly proportional to the complexity of the component.
3	The change in probability of failure of a component with additional testing follows the standard normal cumulative distribution function.

Assumptions 1 and 2 are the basis for the model development and identify our intuition about testing and complexity metrics. Assumption 1 expresses the belief that the reliability of a component increases as the component executes without failure on an increasing number of test cases. Assumption 2 implies that there is a positive correlation between the component complexity and errors.

Assumption 3 reflects how the model assigns a probability of failure to each of the components in the system based on the complexity metrics. A component with a very high complexity value is expected to have a large number of errors. Therefore, high complexity components initially have a high probability of failure. As testing progresses, if the component is executed without failure, confidence in the component grows and, hence, the probability of failure of the component drops. The rate at which the probability of failure drops is initially small, and grows faster with increasing number of error-free test case executions. On the other hand, if the component complexity is low, we do not expect many errors to be found. Such components would initially have a low probability of failure and their probability decreases rapidly with increasing error-free testing.

The cumulative normal distribution curve has the properties which captures these intuitions of component behavior. As shown in Figure 1, the range of probabilities are linearly distributed over a specified RANGE of the curve, with the mean complexity component having an initial probability of failure which is 0.5*SCALE. The SCALE factor ranges from 0 to 2 and has the effect of shifting the RANGE of the probability assignment on the 0-1.0 scale. The quantities RANGE and SCALE are calibration constants in the model must be empirically determined.

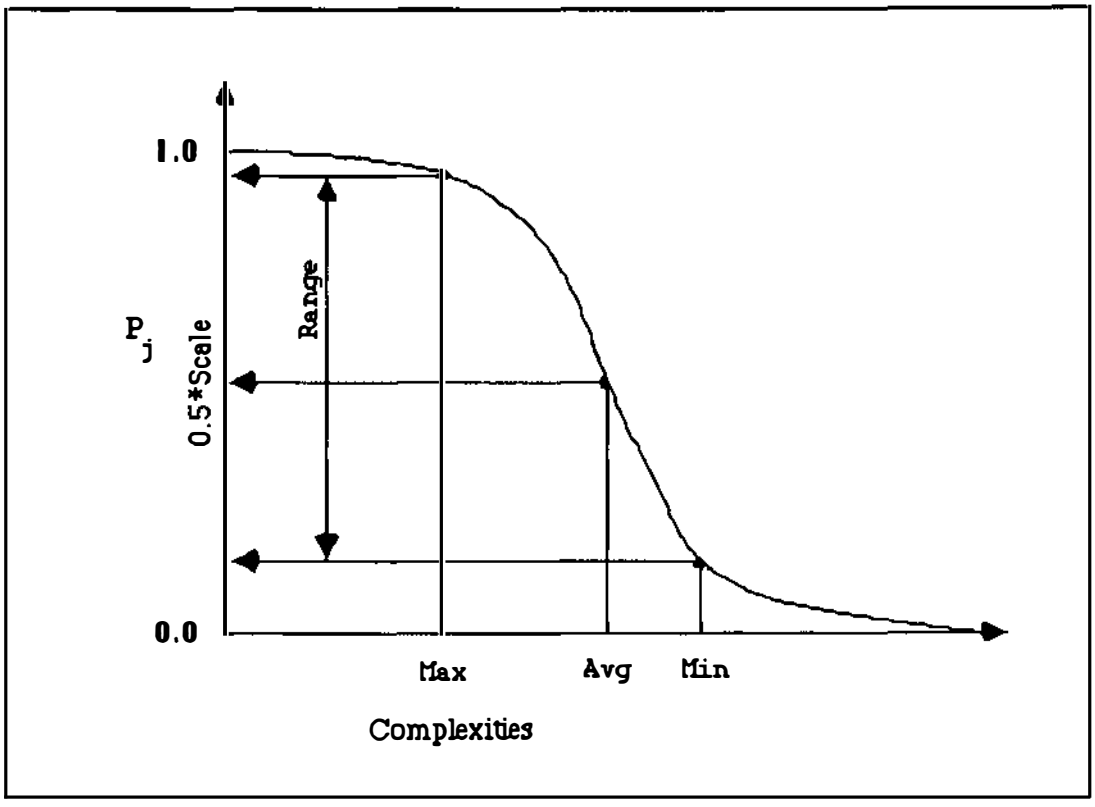


Figure 1: Assigning Initial Failure Probabilities

Empirical observation [Canning 1985] shows that a majority of the errors are discovered in relatively few components and that the majority of these high-defect components are a subset of the high complexity components [Kafura 1985]. Thus, when an error is discovered in a component, its probability of failure should be increased because more errors are likely to be found in this component by additional testing. The increase in probability is determined by the complexity value of the component.

Given the shape of the cumulative normal distribution curve, these effects can be achieved simply by reassigning the component probability value to an ordinate of the abscissa on the left side of the current abscissa. Each time the component is executed without failure, the new probability value is determined to be the ordinate Step test cases to the right of its current position. Whenever an error is discovered in the component, the new probability is the ordinate of the test case that is Back test cases to the left of its current position.

The probability of failure of components change only with testing. This ensures that components which remain untested in the system will retain their assigned probabilities and, thus, if a large portion of the system remains untested, the system probability of failure remains high.

Thus far we have explained the basic assumptions of the model and presented a method of determining the probability of failure of an individual component given its complexity and testing history. From the probability of failure of the components, we can derive the system probability of failure. The notation used in this derivation is summarized in Table 2. The derivation is shown in Table 3.

To determine the frequency of execution of the component we use the behavior of the system we have seen thus far to represent the future behavior of the system.That is, at test case i the degree of testing which a component has undergone is taken to be indicative of the amount of testing the component will undergo in the future. With this assumption the frequency of execution of a component can be expressed as shown in Equation (1) in Table 3. Equation (1) gives the average number of times component j is executed over i test cases.

Table 2: Notation

T	Total number of components in the system.	c_j	Complexity of the jth component.
N	Total number of test cases.	e_{ji}	Total number of errors discovered in component j up to and including test case i.
i	Index over test cases: $1 \le i \le N$.	n_{ji}	Number of test cases in which component j was executed up to and including test case i.
j	Index over components in the system: $1 \le j \le T$.	p_{ji}	Probability of failure of component j after test case i.
P_i	System probability of failure after test case i.		

Table 3: Derivation of the Model

(1) *Frequency of execution of component j*:	$\frac{n_{ji}}{i}$
(2)*Upper bound on failures in component j*:	$(N-i)*\frac{n_{ji}}{i}*p_{ji}$
(3) *Failures over all components in the system*:	$\sum_{j=1}^{T}(N-i)*\frac{n_{ji}}{i}*p_{ji}$
(4) *Number of executions of component j*:	$(N-i)*\frac{n_{ji}}{i}$
(5) *Projected number of executions over all components*:	$\sum_{j=1}^{T}(N-i)*\frac{n_{ji}}{i}$
(6) *System probability of failure*:	$\frac{\sum_{j=1}^{T} n_{ji}*p_{ji}}{\sum_{j=1}^{T} n_{ji}}$

The number of remaining errors in the component is determined by estimating the number of times the component will be tested in the remaining testing period and then applying the probability of failure estimate. This is shown in Equation (2) which is an upper bound on the number of component failures remaining to be discovered. In calculating the upper bound on the number of remaining failures at test case i, we are implicitly assuming that the probability of failure of the component is constant for the remainder of the testing.

Once the projected upper bound on the failures is determined for all the components, we can then determine the projected upper bound on the number of remaining failures in the system. The summation in Equation (3) is over all the components in the system and gives an estimate on the total number of failures remaining to be discovered in the system.

To determine the system probability of failure, we have to estimate the total number of component executions i.e. the sum total of the component executions over all components. This is again a projected estimate based on the testing history of each of the components. The estimate for an individual component is given in Equation (4). This is an estimate on the number of component executions over the remaining test cases. The executions over all components in the system is then the summation shown in Equation (5).

The system probability of failure can then be arrived at as shown in Equation (6). The system probability of failure is taken as the total number of remaining failures over the total number of remaining executions. This is because even if only one component fails, the entire system is considered to have failed.

3. Simulator

In the absence of actual systems with their complexity metrics, testing history, and failure data, we resorted to simulating the characteristics of actual systems and employing error seeding and testing strategies based on random processes.

There are three main functions of the simulator :

1. generate the system descriptions based on specified system characteristics,
2. generate hypothetical error seeding and test case generation information for each system description, and
3. apply the model to each of the system descriptions and their testing history and output the simulation results.

The relationship among these three functions is shown in Figure 2.

The simulator represents a system abstractly as a list of T components

$$< (c_1,e_1), (c_2,e_2), ..., (c_T,e_T) >$$

where the jth components has a given complexity (c_j) and a given number of initial errors (e_j). To make the simulation realistic, the values of the complexities and the number of errors and the pairing of complexity and errors in each component must be carefully controlled. This control is established by constraining the complexities and the number of errors to have a specified mean and variance. Also the pairing of complexities and number of errors is constrained by a specified correlation coefficient. These six parameters (the means, variances, the correlation coefficient, and the number of components in the system) define a system description. As a final means of insuring realism, the values used in the system descriptions are taken from measured production software systems.

It has been observed that the error distribution among components in a system tends to be highly skewed with a few components having a high concentration of errors and a majority with very few or zero errors [Canning 1985]. We found that the bivariate gamma distribution function can be used to model such distributions and also allow us to control the shape and scale of the distribution independently for each of the variates, thus allowing us the flexibility of modelling different system descriptions uniformly.

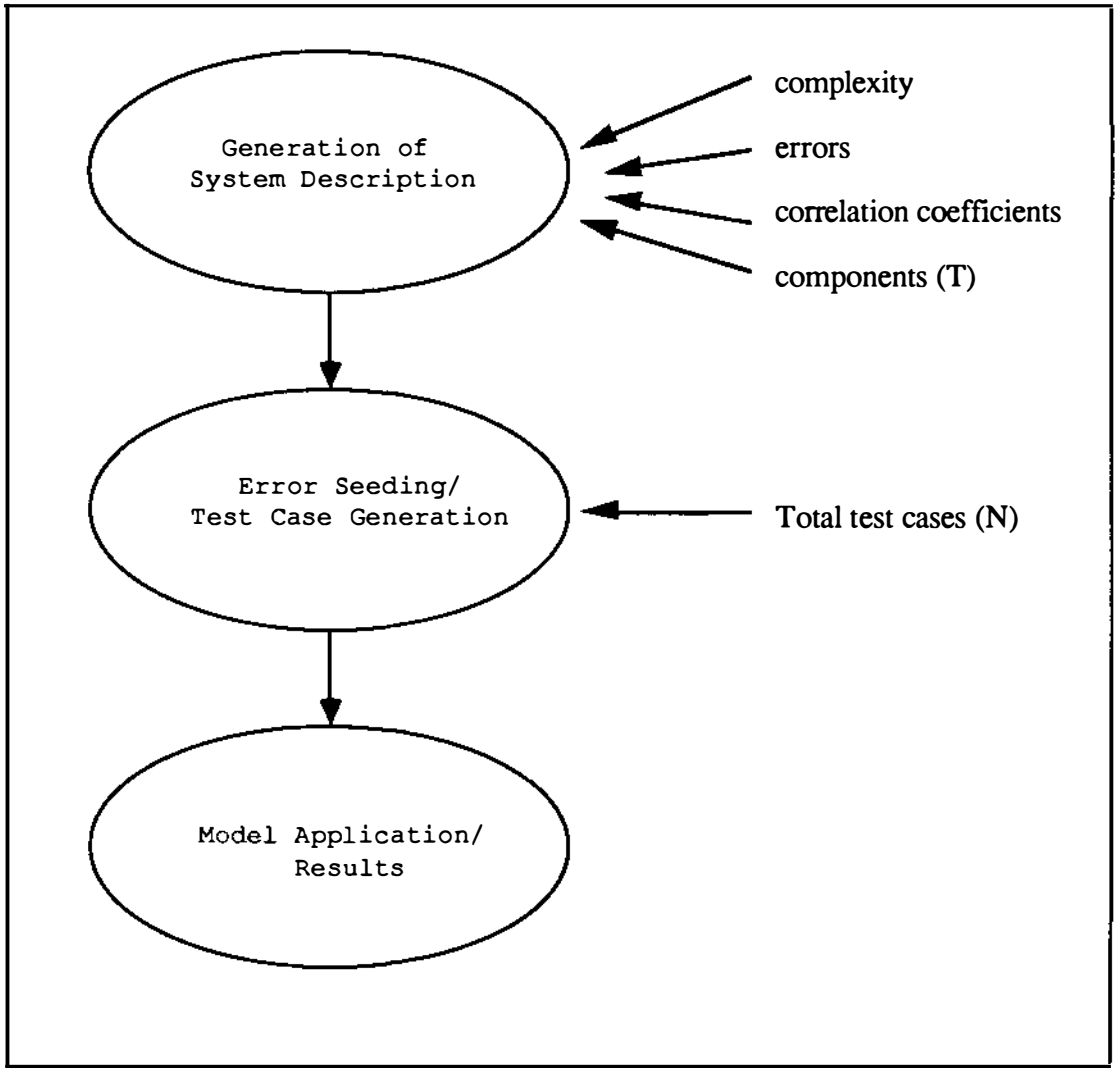


Figure 2: Overall Structure of the Simulator

The methods listed in literature for generating bivariate gamma random variates [Schmeiser 1980], [Schmeiser 1982] produce variates in the real domain. However, we require the variates to be in the integer domain. So, the modified algorithm truncates the random variates generated from the algorithm and recalculates the (mean, variance) parameters for the complexities and the errors in the components. If these parameters fall within the 10% domain of their specified values, the system description is accepted.

The introduction of errors into the code and the relationship between input states and code executed are usually both sufficiently complex processes [Musa 1987] so that modelling testing and error discovery as a random process is a reasonable approach. The simulator constructs testing and error discovery patterns as follows. For each component j in the system, a uniform random number generator is called to determine the number of test cases in which the component is to be executed. This number is then distributed over the test cases by calling a random number generator and linearly mapping the random number over the testing range.The errors, determined in the first step of the simulator, are then distributed over the testing sequence using a uniform random number generator. For each component, j, a random number is generated and mapped over the number of test cases. If the number maps to a test case which has not been marked for error discovery in some other component, the test case information is updated to indicate that component j fails at test case i. The number of errors to be distributed is then decremented.

Once the testing and error distribution information have been determined, the model is ready to be simulated on a test case by test case basis. The simulator generates the probability of failure data using the model developed in Section 2. For comparison, we also compute the actual probability of system failure. The actual probability of failure of the system is defined as the ratio of the number of remaining errors to the number of remaining test cases.

To make a comparison between the actual system probability and the predicted system probability and relative performance of the model over different system descriptions, we generated statistics from the output data of the simulation and performed the Z-tests of significance. We have identified three statistics which could be suitable measures of the degree of closeness of the actual and predicted system failure probabilities :

- Relative difference: the difference between the actual and model estimates of probability (|Pact-Pmodl/Pact),
- 10% Envelope: count of the number of test cases in which the predicted probability of failure is within 10% of the actual probability of failure, and
- 25% Envelope: count of the number of test cases in which the predicted probability of failure is within 25% of the actual probability of failure.

For a given system description a large number of trials were simulated and the above performance measures were calculated for each simulation trail. Finally, the mean value of each measure over the entire set of simulations for a given system description was calculated. The operation of the model was explored by systematically varying the input parameters and observing the effect on the mean performance measures.

To discount for the possibility of statistical anomalies the Z-test of significance was applied to make comparisons of the model performance over different system descriptions. The Z statistic is a normalization of the sample mean differences($\bar{X} - \bar{Y}$). This statistic tests the hypothesis that two population means are the same. In the present context, population constitutes the simulation of a system description. i.e. a population is the test statistics collected over the set of trials for a given system description. The probability of Type I error (probability of rejecting the null hypothesis when it is true) is typically chosen to be 0.1. The boundary value for the Z statistic at this level of confidence is 1.29. A complete description of the hypothesis testing and the Z statistic can be found in any statistics text.

4. Model Evaluation

Iannino, Musa, Okumoto, and Littlewood [Iannino 1984] have also identified several criteria for assessment of any reliability model : predictive validity, capability, quality of assumptions, applicability, and simplicity. *Predictive validity* is the ability of the model to predict future failure behavior from present and past failure behavior. Two additional criteria will be used to assess our model's predictive validity: stability - how sensitive the model is, or is not, to various factors; and directional correctness - does the model operate in an expected manner as the factors are varied. *Capability* refers to the ability of the model to estimate with satisfactory accuracy quantities needed by the software managers, engineers, and users in planning and managing software development projects or running operational software systems. All *assumptions* made in the model development process have been qualified by intuition. The model development presented in the previous section attests to its *simplicity*. It is simple enough to be applied to any software system whose complexity metrics can be generated and can be applied both in the development and operational environment thus making it widely applicable unlike the existing 'black box' models.

To evaluate the model's predictive validity and capability, 20 different system descriptions were generated. As mentioned earlier, a system description is completely defined by six parameters : mean and variance of errors, mean and variance of complexities, total number of components and the correlation coefficient between errors and complexities. Of these parameters only three will be considered here: mean errors, mean complexity and correlation coefficient. The total number of components, fixed in our tests at 250, was not seen as a critical parameter. The variances were studied but, as explained in [Yerneni 1989], the technical limitations in the simulation techniques made these studies inconclusive.

Before considering each of the three factors (mean errors, mean complexity and the correlation coefficient) in detail, we will first consider the summary information given in Tables 4 and 5.

Table 4 shows the range over which each factor was varied and the resulting range of the performance measures. From this summary information we concluded that the model was accurate - its range of relative difference is almost always in the range of 20 - 25% and never more than 54%. This implies that, overall, the model provides good estimates of the actual system reliability.

The stability of the model can also be seen in Table 4. The mean errors is the only factor for which there is a significant variation in the performance measures. The ranges of the other two factors are relatively stable and relatively similar. The sensitivity of the model to the mean number of errors, and how to compensate for this sensitivity, is considered further below.

Table 4: Range of Model Factors and Performance

Factor	Range of Factor	Relative Difference (%)	10% Envelope	25% Envelope
Correlation Coefficient	0.5 - .08	24 - 20	32 - 65	133 - 177
Mean Errors	0.7 - 1.5	22 - 54	1 - 42	8 - 157
Mean Complexity	76 - 114	22 - 24	31 - 45	122 - 149

The last aspect of predictive validity is the directional correctness of the model. To assesses the directional correctness, each factor was varied over the range show in Table 4. The change in performance measures at successive factor levels was observed. Since there were four levels for each factor, there were three differences. In Table 5 the number of differences that are considered directionally correct and the number of these differences which satisfied the Z test are summarized. As can be seen the model is directionally correct in virtually all cases, with the significance also being strong except in the case of the mean complexity factor.

Table 5: Summary of Directional Correctness Results

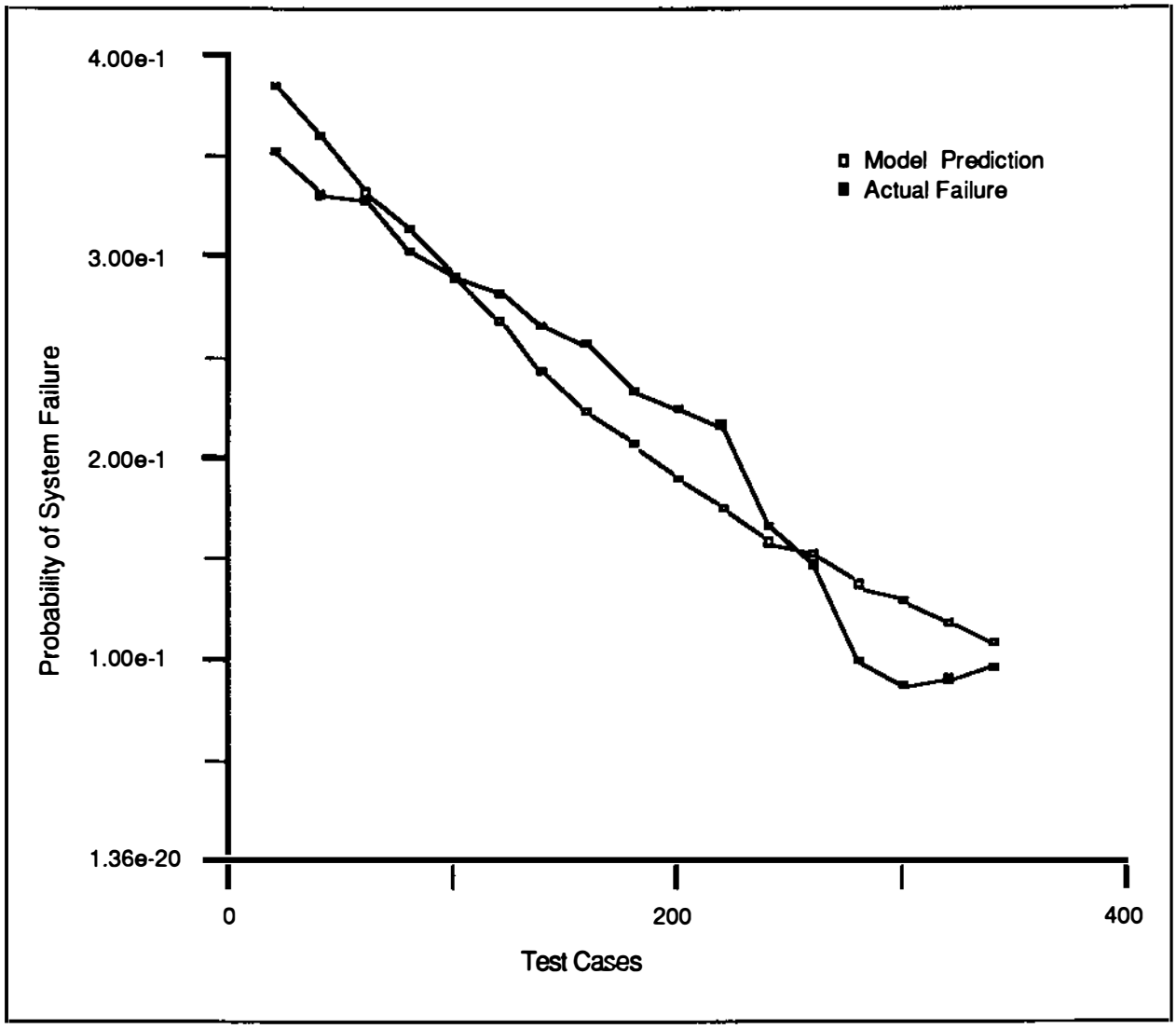
Factor	Directionally Correct			Significance		
	Rel. Diff.	10% Env.	25% Env.	Rel. Diff.	10% Env.	25% Env.
Correlation Coefficient	3	3	3	1	3	2
Mean Errors	3	3	3	3	3	3
Mean Complexity	2	3	2	0	0	2

To illustrate the operation of the model graphically, a single representative simulation is shown in Figure 3. As can be seen there is very close agreement between the two curves representing the model's estimate of reliability and the actual reliability.

Model Behavior with Changing Correlation Coefficient

Table 6 shows the results from performing simulations over system descriptions with varying correlation coefficients. As the correlation coefficient increases, the relative difference between the actual and model probabilities is decreasing and the count of the number of points within the 10% and 25% envelopes in also consistently increasing. Table 7 gives the test statistics for all the three measures. The 10% measure does quite well and passes all the test requirements. However, the 25% measure statistically does not behave consistently. The relative difference measure is not significant in two of the cases.

Figure 3: Model vs. Actual Reliability



We had, however, expected the model to be quite sensitive with respect to the correlation coefficient. One explanation for the insensitivity of the model lies in the orders of magnitude difference in numbers between the complexity values and the errors in the system. Also, an examination of the complexity-error pairs for system descriptions with various correlation coefficients shows that there is a significant overlap in the point spread between different correlations, especially for complexity values in the mid-range. Point spread is the deviation of the errors from the regression line of the system description. This makes it harder for the model to discriminate effectively over the range of correlations.

Table 6: Varying the Correlation Coefficient

		Relative Differences		10% Envelope		25% Envelope	
	Cor. Coeff.	Mean	Variance	Mean	Variance	Mean	Variance
1	.5026	24.62	31.03	32.65	427.89	133.67	2423.19
2	.6056	22.22	40.56	42.63	782.76	157.28	3002.61
3	.7135	21.02	29.03	53.46	1259.18	161.65	2733.45
4	.8125	20.80	45.29	65.26	1155.33	177.22	1839.84

Table 7: Z Statistics from Varying Correlation Coefficient

Differences	Relative Difference	10% Envelope	25% Envelope
1-2	-1.990	2.007	2.241
2-3	-1.006	1.678	0.411
3-4	-0.180	1.680	1.611

One significant problem we have discovered is that the correlation coefficient alone is insufficient to accurately define the relationship between the errors and complexities. Two sets of error-complexity pairs may have the same correlation but there may be significant differences in the variation. Thus, at a given complexity, the deviation could be significant. Most of the system descriptions that we generated have large deviations and thus the output results tend to be inconclusive.

Model Behavior with Changing Mean of Complexities

Table 8 gives the simulation results from varying the mean component complexities. The table does not show any consistent behavior for the relative differences and the 10% Envelope. The tests of significance for these statistical measures in Table 9 also do not show any significant difference. However, the 25% envelope does seem to show that the accuracy of model predictions drops with increasing mean complexities. One reason for this inconclusive behavior is a scattering effect caused by our inability to accurately generate bivariate gamma random variables in the integer domain. We used a modified version of the bivariate gamma random number generator in the real domain. Consequently, we notice a scattering effect in the complexity-error distribution at mid-range complexities. This significantly effects the sensitivity of the model. This effect is more pronounced with higher mean complexities.

Table 8: Varying the Mean Complexity

		Relative Differences		10% Envelope		25% Envelope	
	Complexity	Mean	Variance	Mean	Variance	Mean	Variance
1	76.72	22.55	35.47	45.06	795.28	149.38	2966.80
2	91.44	22.22	40.56	42.63	782.76	157.28	3002.61
3	98.92	23.05	19.49	35.63	851.70	141.00	2090.69
4	114.32	24.53	17.51	31.85	458.93	122.97	1525.24

Table 9: Z Statistics from Varying the Mean Complexity

Differences	Relative Difference	10% Envelope	25% Envelope
1-2	-0.111	-0.428	0.750
2-3	0.704	-1.212	-1.570
3-4	0.243	-0.731	-2.090

Model Behavior with Changing Mean of Errors

The simulation results from varying the mean number of errors in the system is given in Table 10. These results show a significant deterioration in the model's performance with increasing mean error content. One possible cause for this deterioration is that the model was calibrated using system description 1. The rapid deterioration of the model with other system descriptions suggests that the model might have to be recalibrated for a system description with a significantly different mean error content. To test this possibility the simulations were repeated using different model

calibrations. Recalibration can be easily achieved in the model by changing the SCALE parameter. Experimentation revealed that the SCALE parameter must be decreased as the mean error content is increased.

Table 10: Varying the Mean Error Content with Fixed Calibration

		Relative Differences		10% Envelope		25% Envelope	
	Mean Errors	Mean	Variance	Mean	Variance	Mean	Variance
1	0.692	22.22	40.56	42.63	782.76	157.28	3002.61
2	1.020	35.20	26.15	21.34	408.67	52.28	595.99
3	1.224	43.58	25.70	6.59	140.52	27.44	622.28
4	1.552	54.17	22.62	1.14	27.55	8.69	271.23

Table 11 gives the simulation results using a SCALE parameter calibrated for each system description. The results show that the model performs quite well for all the system descriptions after calibrating the SCALE parameter. Therefore, if the mean error content in the system can be estimated by other techniques or by historical experience, the model can be calibrated accordingly to give accurate predictions.

Table 11: Varying the Mean Errors with Adjusted Calibrations

		Relative Differences		10% Envelope		25% Envelope	
	Mean Errors	Mean	Variance	Mean	Variance	Mean	Variance
1	0.692	18.21	73.06	121.36	959.90	200.71	722.32
2	1.020	19.80	69.60	133.49	797.96	189.18	841.41
3	1.224	22.22	62.10	141.81	536.35	178.94	316.87
4	1.552	25.49	64.36	128.33	726.30	170.00	221.06

5. Conclusions

The simulation indicates that the model provides an accurate and stable estimate of actual system reliability. In almost all cases the relative difference measure lies in the 20-25% range. This range of accuracy is surprisingly good in view of the extreme simplicity of the model and the several simplifying assumptions which underlay its construction.

The simulations also show that the model is directionally correct to changes in the mean of the errors. The Z-test shows that the changes are significant. The model also does well with respect to the correlation coefficient albeit below our expectations. The model is directionally correct with varying mean of complexities. However, tests of significance have not been very good.

We also note some limitations of our study. First, the model has only been evaluated on simulations, not on actual systems. Despite our precautions in selecting realistic parameters, any conclusion must be advanced with due caution. However, the encouraging results of the simulations should motivate additional and more realistic evaluation of the model. Second, our inability to accurately generate bivariate gamma random variables in the integer domain effected the sensitivity of the simulation. If a better algorithm can be found and implemented, more accurate comments can be made about the model behavior with respect to the mean of the complexities. Third, we found out that the correlation between errors and complexities alone was insufficient to completely describe a system. There may be other measures which can more accurately capture the relationship between errors and complexities. These measures could then be used in place of correlation in the model. Fourth, the model should be extended to deal with execution time or calender time rather than simply the number of test cases. A comparison of the model's performance with other 'black box' reliability models could then be performed.

References

[Canning 1985] James Canning The Application of Structure and Code Metrics to Large Scale Systems, Ph.D. Thesis, Department of Computer Science, Virginia Tech, May 1985.

[Ehrlich 1990] Willa Ehrlich, S. Keith Lee and Rex Molisani, "Applying Reliability Measurement: A Case Study," IEEE Software, March 1990.

[Henry 1981] Sallie Henry and Dennis Kafura, "Software Structure Metrics Based on Information Flow," IEEE Transactions on Software Engineering, SE-7(5), September 1981, p.510-518.

[Iannino 1984] A. Iannino, J.D. Musa, K. Okumoto and B. Littlewood, "Criteria for Software Reliability Model Comparisons," IEEE Transactions on Software Engineering, SE-10(6), Nobvember, 1984. pp.687-691.

[Kafura 1985] Dennis Kafura and James Canning, "A Validation of Software Metrics Using Many Metrics and Two Resources," Proceedings: Eighth International Conference on Softwar Engineering, August 28-30, 1985, London England, p.378-385.

[McCabe 1976] Thomas McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, SE-2(12), pp. 308-320, December, 1976.

[McClure 1978] Carma McClure, "A Model for Program Complexity Analysis," Proceedings: Third International Conference on Software Engineering, Atlanta, Ga., 1978.

[Musa 1987] John Musa, A. Iannino, K. Okumoto, Software Reliability: Measurement, Prediction, Application, McGraw-Hill, 1987.

[NASA 1982] "Guide to Data Collection" NASA Goddard Space Flight Center, Software Engineering Laboratory Series, SEL-81-101, August, 1982.

[Schmeiser 1980] B.W. Schmeiser and R. Lal, "Squeeze Methods for Generating Gamma Variates," American Statistical Association, p. 679-682, 1980.

[Schmeiser 1982] B.W. Schmeiser and R. Lal, "Bivariate Gamma Random Vectors," Operations Research, Vo. 30, No. 2, March-April 1982.

[Shooman 1976] Martin Shooman, "Structural Models for Software Reliability Prediction," Proceedings: Second International Conference on Software Engineering, pp. 268-280, October, 1976.

[Yerneni 1989] Ashok Yerneni, A Reliability Model Incorporating Software Quality Metrics, M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA 24061, September, 1989.

Project Tracking
It's 1990, Do You Know Where Your Time Is Going?

Katherine Stevens
ADP Dealer Services
2525 S.W. First Avenue, Suite 450
Portland, Oregon 97201
(503) 294-4200

Abstract

One of the earliest software metrics was the percentage of time spent on software development, classified by activity. However, the data processing industry often fails to keep accurate records of project estimates versus actual expenditures.

This paper describes one automated system for recording time and cites some of the benefits offered by a central repository of such information.

One of the fundamental benefits of time tracking and the data it generates is its usefulness as a basis for estimating time requirements for future projects. This assumes that the data on past projects is available, the data is understandable and usable, and future projects are similar to past projects. An automated system helps fulfill the first two conditions by serving as a data repository and providing automatic validation of time entries.

Time tracking also provides a means of evaluating the impact of different methodologies, development environments, and tools on work distribution.

Biography

Katherine Stevens is a Development Manager for the Dealer Services Division of Automatic Data Processing, Inc. (ADP). ADP Dealer Services provides software services to the automotive industry. Katherine has a B.S. in Business Administration from Portland State University. She has fourteen years experience in software development, ten of these with ADP.

Project Tracking
It's 1990, Do You Know Where Your Time Is Going?

Introduction

Tracking time on projects is like exercise; everyone agrees that it's good, but few practice it.

One of the earliest software metrics was the percentage of time spent on software development, classified by activity. However, the data processing industry often fails to keep accurate records of project estimates versus actual expenditures.

This paper describes one automated system for recording time and cites some of the benefits offered by a central repository of such information.

One of the fundamental benefits of time tracking and the data it generates is its usefulness as a basis for estimating time requirements for future projects. This assumes that:

- o the data on past projects is available
- o the data is understandable and usable
- o future projects are similar to past projects

An automated system helps fulfill the first two conditions by serving as a data repository and providing automatic validation of time entries.

Time tracking also provides a means of evaluating the impact of different methodologies, development environments, and tools on work distribution.

Description of the System

For a time reporting system to be effective, employees must understand its importance, must receive some training on how to report time for the work they do, must see the data used in project management and project planning, and, occasionally be strongly encouraged (nagged) to record their time regularly.

An automated system for recording project time should be able to track such basic factors as:

- o time spent on every major component of the project from start to finish
- o time spent on each development activity (analysis, design, code, test, and document)
- o time spent on each classification of work (design specifications, coding walkthroughs, test plans development)
- o time spent by each person on specific tasks (managers want to how who did what on the project)
- o any combination of the above

In Controlling Software Projects, Tom DeMarco suggests a metrics group be used to gather time reports to ensure accuracy and consistency and to eliminate the bias that comes from just putting the time somewhere so that it tallies to forty hours a week. DeMarco also proposes recording only uninterrupted development time in light of his findings that interruptions can delay the resumption of productive work by 25 minutes or more.

In implementing a project time tracking system, we departed from DeMarco's advice by having each developer record time each week and account for all the hours worked. Classifications are provided for time off, sick time, vacations, and administrative, non-project-related tasks. The end result is that overtime is rarely overlooked. When project deadlines necessitate overtime, or are missed because of conflicting demands, employees can be assured that this information is documented. Also, capturing non-project-related time provides the basis for estimating employee availability and controlling these activities.

Auditing time reports is the responsibility of the project lead, supervisor, and a part-time system administrator. Many project leads monitor project progress by having developers submit weekly reports showing time spent on tasks and estimated time to completion. Most supervisors request monthly time reporting summaries.

To standardize reporting, time is posted to defined work classifications. The project description and contents are recorded when the project is established. Time cannot be posted to completed projects. Summaries by activity (analysis, design, code, test, document) are a natural by-product of the work classification structure.

The part-time administrator oversees the maintenance, processing, and reporting functions of the system which now serves over 250 developers. The administrator is responsible for:

- o answering questions
- o training new associates
- o monitoring the system to ensure that time is posted correctly
- o correcting any misapplied time
- o preparing reports for supervisors of associates who are delinquent in reporting
- o adjusting historical data to reflect changes in time reporting standards

The last item provides the ability to refine time reporting standards without risking misinterpretation of historical data collected under one or more sets of different rules.

In the four years that a time reporting system has been in place at ADP, two major overhauls have been made to refine the activity and the work classification breakdowns. The adjustments were made to record changes in development processes and in response to questions generated by analyzing time summaries, such as, "Is the time spent in meetings project-related?"

Trend Analysis

The labor distribution in analysis, design, coding, and testing activities has been well documented for twenty years. During this period, we have seen a shift in development emphasis toward analysis and design activities. In the 1970's, 40 percent of a typical project's budgeted resources went to analysis and design, 20 percent went to programming, and 40 percent was used for integration and testing. In the 1990's, the ratios for some projects are closer to 60 percent analysis and design, 15 percent coding, and 25 percent integration and testing.

Against this background, a single company like ADP can study its own distribution of resources by asking such questions as:

- o Is time being spent in appropriate areas? In appropriate amounts?

- o How does the company compare to industry averages?

- o What is the impact of changes in the development environment such as higher-level programming languages, different compilers, software tools, or PC workstations?

- How is time spent in testing and quality assurance activities throughout the life cycle?

- Are management policies being implemented?

- What percentage of time is spent on non-project-related activities? Is the percentage increasing or decreasing?

These questions provided quantitative feedback of significant importance to ADP in the wake of a major reorganization in 1988. At that time, departments for training, publications, and QA were merged with the programming departments to form product teams. Each team was made responsible for all the development phases of software sets, from project management, analysis, design, coding, testing, to publications and training.

Two major benefits were expected from the reorganization: improved communications and improved quality. The correctness of the decision to reorganize was borne out by subsequent time tracking data. The amount of time spent on project coordination fell by 25 percent in the last two years. In addition, writers and training developers are now able to participate in the project up-front, rather than at the end.

From the data processing literature, one might expect massive changes in the time allocated to activities with the implementation of new technologies, CASE tools, and changes in the development process. Spectacular results may be true for a particular phase of a project. In my experience, however, when the entire development process is considered, a major change may result in a time difference of only a few percent.

The relationship between improved quality and percentages of time by development activity is not straightforward. For example, a development team improved the quality and professionalism of their test plans on a series of successive projects. Yet, the percentage of time spent on test plans and quality assurance activities remained constant. Another project emphasized improving the screen designs to create a more intuitive user-interface. The developers expected to spend more time in the design phase. The product met the design objectives without an increase in the percentage of design time. However, the coding time increased because the user interface was more complex and new utility modules had to be created.

Having comparative data for several years across a wide variety of projects generates as many questions as it answers. Time reporting summaries stimulate an analysis and introspection of projects and processes that is important to continuing improvement.

Estimating

Two fundamental tasks of project management are time estimation and project tracking. An automated project reporting system addresses both by:

- o providing historical project information
- o helping to determine work force availability
- o assisting with project plans to ensure that all necessary tasks are included

Estimating the resources necessary to develop software products is still an inexact science. Researchers have developed several models with varying degrees of promise, but experience with previous, related projects is usually a key ingredient to understanding a project and deriving a viable estimate. Good metrics from past products form the basis for determining how such factors as the number of inputs and outputs, database size, project complexity, staff experience levels, and programming languages will affect its effort and cost.

For historical data to be useful in project estimating, the new project must be reasonably similar to past projects. A consistent methodology, uniformly applied, is essential. Consistent definitions of "design", "unit test", and "system test" must be established and clearly understood. If you don't have one, adopt a project methodology.

A simple spreadsheet can be useful in estimating projects and tracking progress. Project managers can use them to record such attributes as complexity, team experience, number of screens and data elements and correlate them to required development time. Of course, project attributes can also be stored on the automated time reporting system. However, this requires agreement and standardization among project managers as to the attributes and characteristics to be stored.

Critical to the creation of a project schedule is a realistic estimate of resource availability. By recording vacations, holidays, time off, and non-project related (but, hopefully, productive) tasks such as attending meetings and traveling, and interruptions such as fielding questions from other departments, you can determine how much time remains for the actual project.

For example, if employees take two weeks of vacation a year, ten holidays, five sick days, and five days for training, then already 14 percent of the 2,080 hours available in a year are taken. Only 86 percent or 1788.8 hours is available for development projects. At this rate, you would have to budget 114 full-time-equivalent hours for every 100 hours of actual product development.

Monitoring and controlling administrative (non-project-related) time is one way to increase productivity. This implies that you know what these activities are and can track the amount of time spent on them.

Another key element needed for project scheduling is making sure all of the pertinent tasks are included. It's what you didn't plan for that knocks you off track. A project survey by development expert Tom DeMarco found that "unanticipated, unclassified, and miscellaneous activities constituted fully fifteen percent of manpower." A good historical project system allows you to review categories you might have overlooked and to judge the degree to which they might affect your plans.

Conclusion

Creating an automated system for recording time provides an information base for projecting resource requirements, standardizing reporting functions, comparing estimates to actual time, measuring productivity, and monitoring trends.

Every organization will have its own unique requirements for a time tracking system. No system will do everything you want it to do from Day 1, but some benefits will begin immediately. The important thing is to start. Like the Nike ad says, "Just Do It."

References

Barry W. Boehm, *Software Engineering Economics*, Prentice-Hall, Inc., New Jersey, 1981

Tom DeMarco, *Controlling Software Projects*, Yourdon Press, New York, 1982

Capers Jones, *Programming Productivity*, McGraw-Hill, Inc., New York, 1986

Philip W. Metzger, *Managing a Programming Project*, 2nd edition, Prentice-Hall, Inc., 1981

Meilir Page-Jones, *Practical Project Management: Restoring Quality to DP Projects and Systems*, Dorset House Publishing, New York, 1985

"Selling QA", *Systems Development*, July 1989

A SOFTWARE METRICS TOOLKIT: SUPPORT FOR SELECTION, COLLECTION AND ANALYSIS

Shari Lawrence Pfleeger

Joseph C. Fitzgerald, Jr.

Contel Technology Center
15000 Conference Center Drive
PO Box 10814
Chantilly, Virginia 22021-3808
(703)818-4498
pfleeger@ctc.contel.com

ABSTRACT

We describe an approach to metrics tool evaluation that allows a project manager to select a customized set of metrics tools for inclusion in a metrics toolkit. The tool evaluation results are stored in a database organized according to a faceted classification. By providing descriptors of each facet, the manager can retrieve only that tool information that is relevant to the project at hand. Tools can then be compared and contrasted by assigning weights to a set of selection criteria. Finally, selected tools are combined in a toolkit that becomes part of the development environment.

BIOGRAPHIES

Shari Lawrence Pfleeger is head of the Software Metrics Project at the Contel Technology Center's Software Engineering Laboratory. She consults nationwide on software engineering, computer security and other aspects of software systems development. Pfleeger's current research interests involve process modeling, maintenance metrics, and cost estimation. She is the author of research papers in mathematics and computer science, of two university textbooks (*Introduction to Discrete Structures* [Wiley, 1985] and *Software Engineering: The Production of Quality Software* [Macmillan, 1987 and 1991]), and a chapter in Veryard's collection, *The Economics of Information Systems and Technology* [Butterworth, 1990]. She received a B.A. in mathematics from Harpur College, an M.A. in mathematics and an M.S. in planning from The Pennsylvania State University, and a Ph.D. in information technology from George Mason University. Dr. Pfleeger is a member of the ACM, IEEE Computer Society, and Computer Professionals for Social Responsibility.

Joseph C. Fitzgerald, Jr. is a member of the Software Metrics Project at Contel Technology Center, where he does research and development on software metrics. His current assignment includes evaluating software metrics tools and constructing a metrics toolkit for distribution throughout the corporation. On previous projects, he developed two software component library/ retrieval systems to support reuse of existing software. He prototyped other software library systems using retrieval techniques from artificial intelligence. Fitzgerald's research has included investigation of domain analysis methodologies, and the application of research results to the domain of command and control systems. He has over five years of industry experience in computer programming and software engineering. Fitzgerald received his B.S. in computer science from Drexel University in 1987 and is currently pursuing a graduate degree in software engineering at George Mason University. Fitzgerald is a member of the IEEE Computer Society, ACM and ACM SIGAda.

A SOFTWARE METRICS TOOLKIT: SUPPORT FOR SELECTION, COLLECTION AND ANALYSIS

Shari Lawrence Pfleeger
Joseph C. Fitzgerald, Jr.

1. INTRODUCTION

Software Metrics at Contel

Software plays a major role at Contel, from the systems used to assign new telephone services to the tracking and routing done in support of communications satellites. Corporate success depends on the quality of Contel's products and on Contel's ability to respond to its customers in a timely fashion and at a reasonable cost. Thus, management and control of Contel's software development are paramount in assuring that software products are built on time, within budget and in accordance with a stringent set of quality goals.

Software metrics are essential to understanding, managing and controlling the development process. Quantitative characterization of various aspects of development involves a deep understanding of software development activities and their interrelationships. In turn, the measures that result can be used to set goals for productivity and quality and to establish a baseline against which improvements are compared. Measurements examined during development can point to "hot spots" that need further attention, analysis or testing. Even during maintenance, metrics can reflect the effects of changes in size, complexity and maintainability. Measurement also supports planning, as projections and predictions about future projects can be made based on data collected from past projects. Tools and strategies can be evaluated, and the development process and environment can be tailored to the situation at hand.

However, both management and software engineers often balk at the additional time and labor needed to support metrics data collection and analysis. For example, the Software Engineering Laboratory at the University of Maryland reports that data collection and analysis add 7 to 8% to the cost of a project [Card90], and DeMarco estimates that development costs increase between 5 and 10% when metrics collection is involved [DeMarco90]. Thus, tools and techniques must be available to software engineers to minimize the degree to which they are distracted by their metrics duties. Only then will metrics become a welcome part of software development.

At the Contel Technology Center, we are addressing this need in several ways. As part of the Process and Metrics Project, the Software Metrics Program provides services and products to facilitate the use of metrics throughout the corporation. We recommend that the collection of metrics be tied to the maturity of the development process. As discussed in [Pfleeger89] and [Pfleeger90], a project can measure only what is visible and appropriate. Thus, an immature project (where requirements are not well-known nor understood) begins by measuring effort and time, so that a baseline can be established against which improvements can be compared. Next, when requirements are defined and structured, project management metrics establish general productivity measures. When the process is well-defined enough to support it, a project adds product measurement; in this way, characteristics of intermediate products can be used to suggest the likely quality of the final product. If the project is mature enough to have a central point of

control, then process measures with feedback to the controller or manager are appropriate. The feedback is used to make decisions about how to proceed at critical points in the process. Finally, the most mature projects can use process measures and feedback to change the process dynamically as development progresses.

Notice that the process maturity framework suggests metrics not only to monitor the activities in the development process but also to help improve the process itself. By considering metrics as nested sets of measures related to process maturity, each level allows management more insight into and control over the process and its constituent products.

To support metrics collection and analysis, we provide each development project with a software metrics toolkit tailored to individual project needs. The toolkit contains metrics tools to collect and analyze data appropriate for the project's process maturity, development environment, and management needs and preferences. Underlying the toolkit is a project metrics database. The database supports two activities. First, its contents can be used by the tools and by project managers to monitor and make decisions about the development process. Second, the database contents can be transferred to a Contel corporate historical database. The corporate database is analyzed to examine trends, make predictions, and set standards for future projects.

The remainder of this paper focuses on the development of the metrics toolkit. We describe the way in which we evaluate existing tools, and the methods we use to make the evaluations available to Contel's project managers.

Evaluation Philosophy

Many automated metrics tools are available today. They vary from simple analyzers that count the number of decision points in a source listing to complex estimators of cost and schedule. Some tools are stand-alone, while others are embedded as part of a suite of computer-aided software engineering (CASE) tools. We have chosen to evaluate any tool that involves metrics data collection and analysis, whether commercially available or provided free to interested users. To do so, we have developed a two-stage approach to evaluation based on the work of [Bohner89]. The first stage, called a *paper evaluation*, reviews the product literature and documentation to determine

- the intent of the tool
- the type(s) of metrics supported by the tool
- the environment in which the tool is to be used
- the available interfaces between this tool and other tools
- the type of user interface provided by the tool

The results of the paper evaluation are used to suggest a small subset of metrics tools that deserve further investigation, based on our needs at Contel. In particular, we choose candidate tools for inclusion in a metrics toolkit; the toolkit is then built by the Process and Metrics Project to assist software engineers in implementing the recommended metrics.

Once the candidate tools are selected for more detailed analysis, the actual tool (rather than its paper description) is installed and evaluated against a set of criteria. This second stage, called the *extended evaluation*, investigates each tool in depth. Tool characteristics such as the following are examined:

- speed
- data import and export capabilities
- quality of the user interface
- documentation
- tool accuracy
- vendor support
- cost

The paper and extended evaluations provide enough information to allow project managers to make informed decisions about the tools they wish to include in their project's development environment.

The paper evaluation is addressed in Section 2. First, the general framework is discussed. Then, each characteristic is described in turn. Finally, the paper evaluation is illustrated with an example. Section 3 discusses the extended evaluation. We display a rating form, showing the items included in the evaluation and the possible sources of information for each item. An example explains how the extended evaluation is performed.

We describe an example of the use of our evaluations in Section 4. In addition, we explain how project managers have on-line access to the evaluations stored in an Oracle database. Updates and additions will be made as existing tools change and new ones are introduced. Section 5 describes the relevance of the metrics toolkits and tool evaluations to the on-going work of the Process and Metrics Project. Finally, we draw conclusions about the types of metrics available for use today, their price, ease of use, and utility at Contel.

2. PAPER EVALUATION

Framework

The choice of metrics to be collected on a project is determined by the maturity of its software development process, the availability of data, and the needs of project management. To assist in the collection of metrics throughout the corporation, the Software Metrics Program develops a toolkit for each project, based on the metrics needs and the results of our evaluations. A project manager specifies information about the project (e.g. its environment, methods, and metrics needs), and the evaluation database is used to suggest appropriate metrics tools.

The continuing tools survey gives us an understanding of industry's ability to provide corporations with useful metrics tools. In discovering which tools need improvement, we can direct vendors towards making those improvements. Moreover, in collecting and storing tool information in a database, Contel software managers can locate tools without having to perform individual evaluations themselves.

The first step in the evaluation process gathers information. Candidate tools are considered for inclusion in our toolkit. Information available on each tool is collected from the vendors, and existing third party evaluations are sought. From the collection of information available for each tool, we perform a paper evaluation. The evaluation classifies the tool according to certain characteristics, described below, so that the entire body of tool evaluations forms a repository of

tools information. Project managers can then query the repository to determine types of tools available, functionality, cost, and any other relevant information.

The results of the paper evaluation are recorded at the top of a tool evaluation form, shown as Figure 1. The top of the form captures the name of the tool, the vendor, and the date of the evaluation.

Metrics Tool Evaluation Report

1.0 The Tool

Tool Name:	Ada Metrics Analysis Tool (AdaMAT)
Vendor Name:	Dynamics Research Corporation
Vendor Address:	60 Frontage Rd
	Andover, MA 01810
Contact/Phone #:	John Ragasta or John Rice/ 508-475-9090
Evaluation Date:	05/21/90

Figure 1: Tool Evaluation Identification

Tool Classification

The next section of the evaluation classifies the tool according to several criteria. The classification scheme follows a methodology known as *faceted classification.* Facets are multiple independent indices used to identify groups of similar objects. That is, each facet characterizes an attribute of the object that cannot be described using any of the other facets. In this sense, facets are orthogonal. Such a scheme has been very successful in classifying components for a reuse repository [Prieto-Diaz87].

The classification scheme allows tool information to be stored and retrieved based on the interests and needs of the querying project manager. The indices or facets we have chosen are:

1) **Type**: the type or purpose of the tool, such as cost estimation, code analysis, etc.

2) **Activity**: the development step to which this tool can be applied, such as requirements, design, code, test, etc.

3) **Level**: the minimum level of process maturity for which this tool may be appropriate

4) **Method**: the development method or technique supported by the type of tool being described (such as COCOMO, SADT, or Jackson System Development)

5) **Language**: language dependencies of the tool (e.g., a code analyzer might parse Ada and C, but not COBOL)

6) **Operating System**: the operating system required for the tool to run

7) **Platform**: the hardware platform required for the tool to run, such as IBM PC or VAX

8) **Target Application**: the system type that the tool is designed for, such as "real time" or "MIS"

Each facet descriptor is entered in Section 2 of the evaluation form, as shown in Figure 2. The entries for each descriptor are derived from vendor literature, discussions with vendor representatives, and/or evaluations reported in journals and trade publications. The facet characterization allows us to describe every situation in which the tool can be used. Thus, the form accommodates multiple descriptors for each facet. For example, if a tool runs on several platforms, under several operating systems, or with several design techniques, the facet descriptors can reflect these situations. Notice that Figure 2 indicates that AdaMAT runs both on a Vax and on a Rational platform. Similarly, AdaMAT can provide metrics for coding, testing or maintenance.

A major benefit of faceted classification is the ease with which additional facets can be added to the scheme. Unlike hierarchical or other database structures, a faceted approach allows us to keep intact the existing information and merely add the additional information needed to describe the new facet. The only restriction on new facets is that they be independent of existing facets. In other words, the aspect of the tool characterized by the new facet cannot be described by using a combination of descriptors with the old facets.

Metrics Tool Evaluation Report

1.0 The Tool

Tool Name:	Ada Metrics Analysis Tool (AdaMAT)
Vendor Name:	Dynamics Research Corporation
Vendor Address:	60 Frontage Rd
	Andover, MA 01810
Contact/Phone #:	John Ragasta or John Rice/ 508-475-9090
Evaluation Date:	05/21/90

2.0 Tool Classification

Type	Activity	Level	Method	Language	OS	Platform	Target App.
analysis	code	2		Ada	R1000	Rational 100, 300C	
	maintenance				VMS	Vax	
	test						

Figure 2: Tool Classification

The tools evaluation information is stored in an Oracle database according to its faceted classification. Queries made in terms of facets allow project managers to read only those tool evaluations that suit their needs. For example, a manager of a project that is written in Ada on a Rational platform may need a tool that will generate complexity metrics for the project's code. Here, the manager will set Type to "complexity", Language to "Ada", and Platform to "Rational", and the result will be a list of tools that analyze complexity for Ada programs in the specified

environment. The manager can be more specific, asking for instance for a Method of "cyclomatic complexity" to get only those tools that implement McCabe's metric in the desired environment.

The top two sections of the evaluation report represent the information gathered in the paper evaluation step. The third section is used only for the extended evaluations.

3. EXTENDED EVALUATION

Evaluation Criteria

The extended evaluation of a tool involves the use of a tool in a real-life setting. That is, a hands-on evaluation is done using Contel data with a functioning version of the tool (not a demonstration copy). The results of the evaluation are recorded in Section 3 of the evaluation form, as shown in Figure 3. The first portion of this section contains information on the version, platform and operating system of the tool as used in the hands-on evaluation. This information is used to prevent inconsistencies in evaluation data across different platforms or as vendors release new versions of their tools.

Next are subjective evaluations of the tool's strengths and weaknesses; this part of the extended evaluation documents those issues that are not addressed in the more objective rating scheme that follows. The final portion of Section 3 contains a summary table of the tool's objective evaluation. The criteria for evaluation fall into seven categories:

1) **Performance/Speed**: rates the execution of the tool in performing its calculations or analysis

2) **Data Import/Export**: refers to any means that the tool provides for importing/exporting data from/to other tools (the simpler data transfer mechanisms get higher scores)

3) **User Interface**: refers to ease of use and ease of learning the tool, so that it can be used effectively in an organization

4) **Documentation**: refers to the availability and overall quality of the documentation provided with the tool

5) **Tool Accuracy**: rating given to judge a tool's accuracy in implementing a model for a certain metric (e.g., COCOMO for size estimation) and the tool's flexibility in providing modifiable parameters to the model it implements

6) **Vendor Support**: rating on the vendor's provision of support (e.g. an 800 number or help line), response time in answering questions, and the helpfulness of the information provided

7) **Cost**: criterion based on the cost for the corporation to use this tool on a company-wide scale, not just the single license fee.

1.0 The Tool

Tool Name:	Ada Metrics Analysis Tool (AdaMAT)
Vendor Name:	Dynamics Research Corporation
Vendor Address:	60 Frontage Rd
	Andover, MA 01810
Contact/Phone #:	John Ragasta or John Rice/ 508-475-9090
Evaluation Date:	05/21/90

2.0 Tool Classification

Type	Activity	Level	Method	Language	OS	Platform	Target App.
analysis	code	2		Ada	R1000	Rational 100, 300C	
	maintenance				VMS	Vax	
	test						

3.0 Tool Evaluation

Version:	1.0
Platform:	Rational 100
Op. System:	R1000
Cost:	Rational: $20,000
	Vax: range from $5,000 to $25,000 depending on platform
Strengths:	Provides written feedback on potential problem areas in code, applies real software engineering principles (those that are both independent of and dependent on the Ada language itself)
Weaknesses:	price; not available on PC platform as yet.

TABLE 1.

Criteria	Raw Score (0-10)	Weight (0-10)	Total
Performance/Speed	can't objectively evaluate on heavily loaded Rational		
Data Import/Export	4		
User Interface	7 (consistent with Rational)		
Documentation	5		
Tool Accuracy	9		
Vendor Support	8		
Cost	6		

Figure 3: Results of Extended Evaluation

The criteria are arranged in tabular form, as shown. The left column contains the raw score the tool received in the evaluation, ranked from 0 (low) to 10 (high). The second column represents a weight given to each of the criteria. This weight is assigned by the project manager doing the evaluation. That is, each project will have different needs and goals, and therefore different desirable characteristics.

Evaluation Tool and Database

The evaluation database holds the results of all tool evaluations. A tool built on the database allows the project manager to tailor the evaluation to the needs of the project at hand, as shown in the example in Section 4. The final score for each criterion is computed by multiplying the raw score by the assigned weights; if desired, the scores can be summed to yield a single number that represents the tool's overall rating. In this way, ratings can be compared and contrasted by the project manager and used to support final tool selection.

4. BUILDING A TOOLKIT: AN EXAMPLE

The Software Metrics Program team uses the evaluation results to build a metrics toolkit tailored to a project's needs. To see how the toolkit is built, we present an example. Suppose Manager A is beginning a software development project. According to the requirements, a system is to be developed using Objective C on a Sun workstation. Manager A consults with the Software Metrics Program team and decides that development will be at maturity level 3. According to the nested levels of metrics recommended, Manager A plans to collect data for project management and project metrics. In particular, tools in the coding phase are required to analyze Objective C. The manager specifies queries by supplying descriptors for each facet of interest. For instance, the Language is Objective C, the Platform is a Sun, and the Activity is coding. The database responds with all tools described by these characteristics. Because Manager A is concerned about software reliability, it may be important to know the complexity of intermediate products. The database can be queried for complexity tools in particular by specifying Type as complexity, along with the other facet descriptors.

Next, Manager A reviews the evaluations of tools suggested as a result of the queries. It is up to the manager to select tools that are most appropriate for the needs of the project. For example, the database may suggest a code analysis tool that reports a variety of Halstead metrics. However, the manager may decide that there is no need for such measurements on this project. Similarly, the manager must decide whether to use a tool that runs in the Sun environment with limited capability, rather than one that runs on another platform with increased capability.

Manager A is given the opportunity to weight each tool according to the project's priorities. For example, suppose that Manager A is evaluating Metriscope, a tool to compute code complexity. Performance is very important to the project manager, because the tool will be used repeatedly by many programmers on very large code modules. The raw scores for Metriscope are listed in the first column, and Manager A adds weights in the middle column, as shown in Figure 4. The totals for each criterion are calculated automatically.

1.0 Tool Classification

TABLE 1. Metriscope

Criteria	Raw Score (0-10)	Weight (0-10)	Total
Performance/Speed	5	10	50
Data Import/Export	4	2	8
User Interface	5	1	5
Documentation	5	1	5
Tool Accuracy	9	2	18

Figure 4: Weighting of Metriscope Tool by Manager A

By comparison, the same weights for the CompleMetricks tool yield the scores shown in Figure 5. It is important that Manager A examine the scores for each criterion, not just the total score. It is clear that, on the basis of performance and speed, Metriscope is the tool of choice. However, several criteria may be important, and the manager must take each into account when making a final decision.

TABLE 1. CompleMetricks

Criteria	Raw Score (0-10)	Weight (0-10)	Total
Performance/Speed	2	10	20
Data Import/Export	3	2	6
User Interface	5	1	5
Documentation	5	1	5
Tool Accuracy	10	2	20

Figure 5: Weighting of CompleMetricks Tool by Manager A

Similarly, Manager B may examine the same tools but with different priorities. For example, Manager A may have rated user interface low because the software engineers are familiar with metrics and the use of metrics tools. However, Manager B's employees are using metrics for the first time, so user interface and documentation are very important.

TABLE 1. Metriscope

Criteria	Raw Score (0-10)	Weight (0-10)	Total
Performance/Speed	5	1	5
Data Import/Export	4	2	8
User Interface	5	10	50
Documentation	5	8	40
Tool Accuracy	9	1	9

Figure 6: Weighting of MetriscopeTool by Manager B

In this way, tools can be compared and contrasted. The final decision on tool use is always left to the project manager.

Once the set of tools is chosen, their interaction with the development environment is considered. If the full complement of tools needed can run on a Sun workstation, then the Software Metrics team works with Manager A to buy and use the tools on the Sun. Help is given in establishing a project historical database on the Sun, and the tools are supplemented with special-purpose code, if necessary, to populate the database with data collected. Finally, the Metrics team helps Manager A interpret the data and transfer the results to the Contel corporate database for archival purposes.

However, some of the types of tools recommended may not be available for the Sun. In this case, the Metrics team identifies tools that run on an IBM PC and that meet Manager A's requirements. The project historical database is built on the PC, and the PC is used to supplement the on-going development on the Sun workstations. In this case, the data is transferred from the PC to the corporate database at project end. The PC is chosen as a platform for the tools because of its moderate cost and the abundance of metrics tools available to run on it.

Updates and additions will be made as existing tools change and new ones are introduced. In addition, users of tools will be interviewed, and ratings changed to reflect experience with each tool. The strengths and weaknesses section of the tool evaluation report is likely to grow as user experiences are captured there.

5. CONCLUSIONS

The metrics tool evaluations that we have performed have revealed only a limited number and kind of metrics tools available in the marketplace. Two general categories describe almost all of the tools on the market:

- project management and cost estimation tools
- code analysis and testing tools

In this sense, our work has identified several major areas in which other metrics tools are needed:

- requirements-related metrics tools (for complexity, completeness, uncertainty, etc.)
- process-related metrics tools
- maintainability metrics tools, including impact analysis tools

Moreover, the tools that exist vary widely in their functionality, sophistication of user interface, and price. The greatest lack of today's metrics tools is their inability to be integrated easily with one another or with the development environment in which they are placed. However, in spite of their limitations, the existing tools provide invaluable information about the way we develop software.

The goal of Contel's Software Metrics Program is to make metrics collection and analysis a natural and helpful part of the software development process. To that end, the software metrics evaluation database puts metrics tools information at the fingertips of those who need it. The resulting toolkits enable managers to integrate measurement into the development activities. In addition to its use for forecasting and scheduling, measurement is invaluable for providing the visibility into the process needed for informed decision-making about software development products and activities.

Moreover, the measures provide a baseline against which improvement can be measured, so that goals for productivity and quality can be set and attained.

The Software Metrics Program will continue to evaluate and track the use of metrics tools throughout the corporation. Future plans include the development of analysis tools that, in concert with our corporate metrics database, will allow us to make decisions about metrics and metrics tools based on past successes. We plan to describe characteristics of the development process itself and incorporate those descriptors as facets in our database. In this way, tools will be selected not only based on development environment and tool strengths and weaknesses, but also on the type of development process or process activities.

6. REFERENCES

[Bohner89] Shawn Bohner, *Computer Aided Software Engineering Tools Evaluation Criteria*, CTC-TR-89-008, Contel Technology Center, 1989.

[Card90] David Card and Robert Glass, *Measuring Software Design Quality*, Addison-Wesley, 1990.

[DeMarco90] Tom DeMarco, Comments made at the 12th International Conference on Software Engineering, Nice, France, March 1990.

[Pfleeger89] Shari Lawrence Pfleeger, *Recommendations for an Initial Set of Software Metrics*, CTC-TR-89-017, Contel Technology Center, 1989.

[Pfleeger90] Shari Lawrence Pfleeger and Clement L. McGowan, "Software Metrics in a Process Maturity Framework", *Journal of Systems and Software*, July 1990.

[Prieto-Diaz87] Ruben Prieto-Diaz and Peter Freeman, "Classifying Software for Reusability", *IEEE Software*, 1987.

Project Management

in a

Prototyping Environment

by William H. Roetzheim

Project Management in a Prototyping Environment

This talk is based on my books:

Structured Computer Project Management (Prentice-Hall)
Structured Design Using HIPO-II (Prentice-Hall)

Levels of Prototyping

1. Dialogue
2. Data entry
3. Report
4. Database
5. Calculations and logic
6. Applications packages
7. Rapid

Dialogue Prototypes

Purpose:
Develop and group functional requirements.
Map top level implementation scheme.
Provide framework for prioritizing requirements.

Used by:
Analyst to solicite feedback.
End user to provide input.
Project manager for function based estimates and plans.
Project manager plus customer for design to cost.

Useful during:
Requirements analysis.
Later stages as a stub to other prototyping tools.

Abuses:
Customer design treated as final.
Analyst design treated as final.
No cost factor tied to options.

Report Prototypes

Purpose:
Database design.
Functional requirements analysis.
Dataflow analysis.

Used by:
Customer's managers to provide input.

Useful during:
Early design.

Abuses:
Artificially short reports.
Failure to prioritize reports.

Data Entry Prototypes

Purpose:
Validate/demonstrate screen layout.
Efficiency testing.
Performance requirements validation.
Database design input.

Used by:
Data entry personnel to provide input.
Customer to generate cost/benefit information.

Useful during:
Early design.

Abuses:
Brief demos/trials.
Unrealistic response times.

Database Prototypes

Purpose:
Demonstrate/test interactive queries.
Demonstrate/test custom report capabilities.
Design database schemas.

Used by:
Analysts to develop schemas.
Customer's managers to provide input.

Useful during:
Early to mid-design.

Abuses:
Artificially small data sets.

Calculations and Logic Prototypes

Purpose: Test complex algorithms.
Timing analysis.
Accuracy analysis.

Used by: Lead engineers.

Useful during: Early development.

Abuses: Invalid simplifications.
Over generalizations.

Rapid Prototypes

Purpose: Incremental development.
Risk reduction.
Requirements refinement.
Iterative costing.

Used by: All.

Useful during: All stages.

Abuses: Re-use of code from previous iteration.

Applications Package Prototypes

Purpose: Educate first time users.

Used by: Analysts.

Useful during: Requirements definition.

Abuses: Assumption that application package capabilities are the starting point.

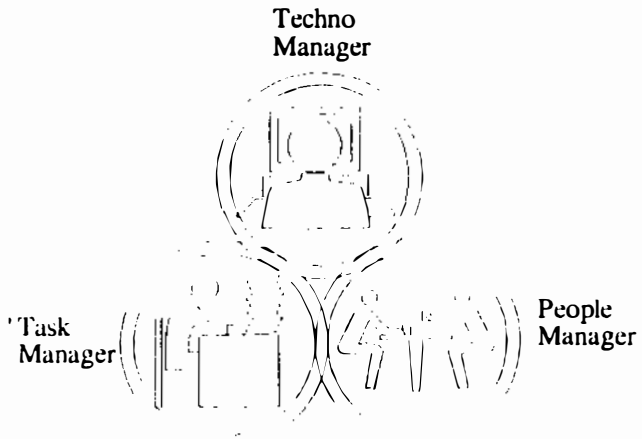
Rapid Prototyping Advantages

- Improved code maintainability if done right.
- Software better meets user needs.
- Lower risk development.
- Accurate reflection of real world limitations.

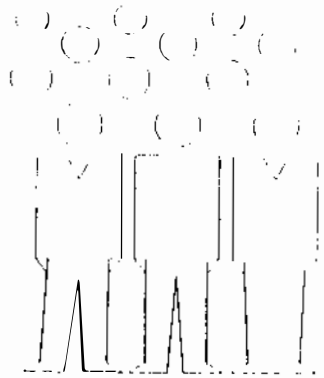
Rapid Prototyping Disadvantages

- Decreased code maintainability if done wrong.
- No good idea of up-front cost.
- Difficult to get approval for software development.
- Difficult to evaluate proposals.
- Employee turnover can cause major problems.

Types of project managers



Techno-Manager view of people



Techno-Manager traits

Does...	Has...
Focus on project management science.	Multiple versions of project management software, including Primavara
View people as resources.	A large size plotter for producing wall size CPM plots.
See projects in terms of milestones.	One (or more) certificates in project management.
	Lots of money.
	Not much fun.

Strengths versus Weaknesses

Task Manager view of people

Task Manager traits

Does...	Has...
Work 12 hour days, 6 days per week.	Minimal project management tools.
Distrust employees.	An epson printer for output.
Focus on work to be done now.	No formal training in project management.
See projects in terms of looming deadlines.	Some money.
	No fun.

Strengths versus Weaknesses

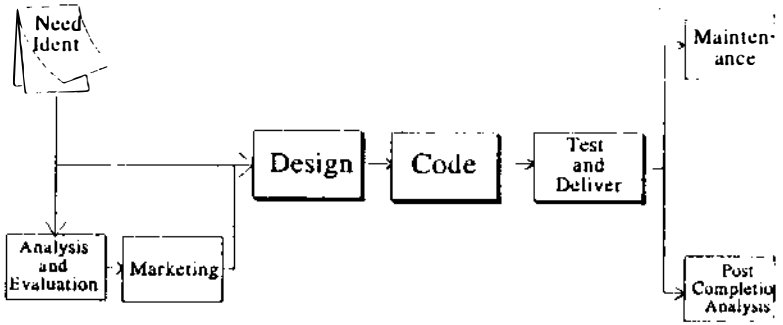
People Manager view of people

People Manager traits

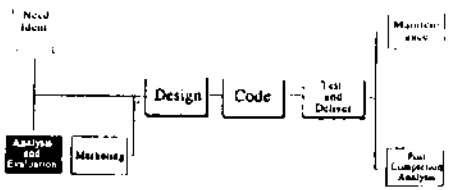
Does...	Has...
See the project in terms of people (customers, team, management, etc.)	MacProject
Facilitate work.	2 copies of Peopleware.
Guide team motivation and understanding.	Childrens drawings on his office walls (rather than CPM plots).
Partition tasks for optimum employee productivity.	Some money.
	Lots of fun.

Strengths versus Weaknesses

Project Management



Expected Value = Return × Probability of success



Expected Value = Return × Probability of success

Analysis and Evaluation

Risk Analysis

Risk Areas

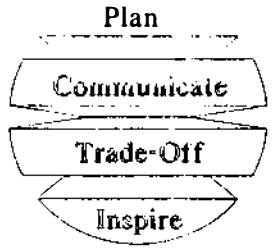
- Technical
- Cost
- Schedule
- Network

Risk Factors

- Likelihood of Failure
- Consequence of Failure
- Overall

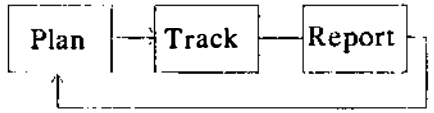
You market to:

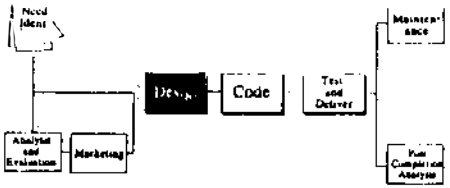
Your company
Your project team
External participants (e.g. banks)
The customer



You market by:

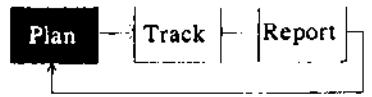
Preparing a good plan
Maximizing the benefits to each group
Minimizing the risks to each group
Packaging your ideas in a technical proposal

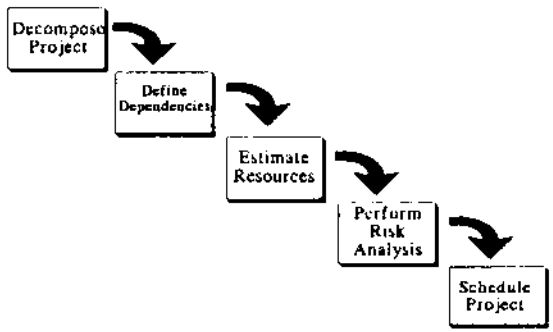




Project managers should be active participants in the program design effort:

- Planning
- Communicating
- Performing trade-off analysis
- Inspiring





Three Stages of Planning

1. Concept oriented
2. Function oriented
3. Implementation oriented

HIPO-II Classes of Program Modules

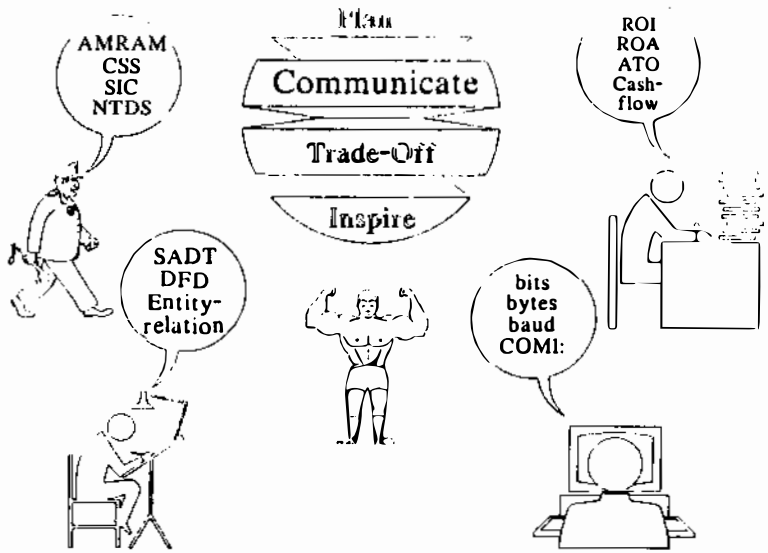
1. Menu
2. Interrupt
3. Keyboard
4. Processing
5. Common
6. Library

Effective tracking is:

- Exception oriented
- Sufficiently standardized to allow comparisons
- Future oriented

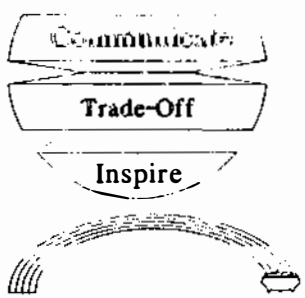
Effective tracking requires:

- Project team communication
- Tracking project resources
- Monitoring project schedule



Trade-Off

- Trade-off decisions are made daily during the project.
- Most are of limited scope and never involve the customer.
- All have a cumulative effect on cost and performance.
- Your plan must support effective trade-off analysis.

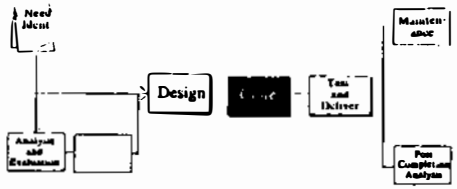


It is your job to lead/inspire the design team. You must instill the customer's vision of the software into your team.

The critical concept of baseline design

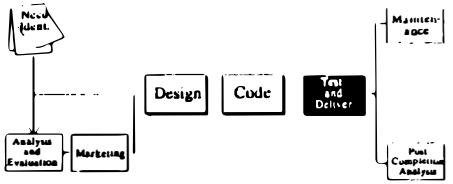
1. For a typical application, 20% of the code will deliver 80% of the functionality.

2. Be sure to deliver that key 20%!



Project Manager's Job

- Coach the programming team
- Ensure quality of code
- Develop your programmers
- Shield your programming team
- Work with external individuals
- Work to expand scope



Project Manager Responsibilities

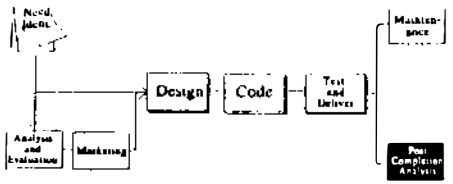
Quality Assurance
- Quality of deliverables
- Ensuring contract requirements are met

Selling delivered solution

Advanced Techniques To Be Aware Of

Cost Estimation

General
Top down versus Bottom up



Project Manager Responsibilities

- Plan for the end
- Prepare lessons learned report
- Evaluate project team
- Marketing opportunities report

Advanced Techniques To Be Aware Of

Cost Estimation

Models
- Informal
- Cost allocation approach
- LOC based
- Function points based

Advanced Techniques To Be Aware Of

Cost Estimation

Stages
- Concept
- Requirements (or function)
- Implementation

Advanced Techniques To Be Aware Of

Scheduling

1. Prepare WBS
   - Hierarchical
   - Product oriented
   - Includes all deliverables

Advanced Techniques To Be Aware Of

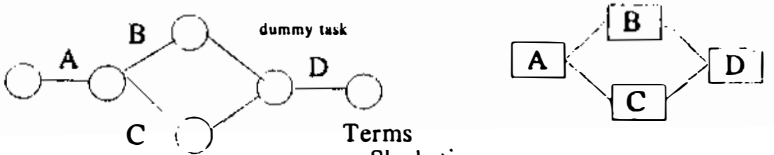
Scheduling

2. Estimate duration and standard deviation
   - Estimate best, expected, and worst case
   - Mean = [best + (4 * expected) + worst] / 6
   - SD = (worst - best) / 3.2

Advanced Techniques To Be Aware Of

Scheduling

3. Estimate dependencies



Terms
- Slack time
- Critical path
- Successor task
- Predecessor task
- Link
- Link type

A Survey of Test Tools for Toolsmith and Users

Dr. Boris Beizer

1232 Glenbrook Road
Huntingdon Valley, PA 19006

Abstract

Test tool taxonomy, terminology, and features: commercial, research, and private tools; comprehensive test: environmental future evolution; toolsmithing budgets; and the tool penetration problem.

Biographical Sketch

Dr. Boris Beizer is best known for his books on testing and his seminars based on those books: Software Testing Techniques and Software Systems Testing and Quality Assurance. He received a Ph.D. in Computer Science from the University of Pennsylvania and has been in the computer industry for more than 30 years. As an independent consultant, he works with leading software development organizations on software testing and quality assurance problems.

Regression Testing Tools for Embedded Software Control Systems

Ron Blair, Dan Uhrich

Honeywell Inc.
1000 Boone Ave. N., Golden Valley, MN 55427
(612) 541-7749, (612) 542-6841
uunet!blair@hi-csc.honeywell.com
uunet!fumus!uhrich@src.honeywell.com

Abstract

The task of retesting software is labor intensive and repetitious. When performed manually, the process is error prone, uninteresting and costly, yet vital, as a mechanism for identifying and fixing software bugs and ascertaining product quality. The retesting problem is exacerbated in control systems products because the software is often subjected to multiple code compression steps added to the development life cycle.

The solution for Honeywell's thermostat controller product line is a regression testing tool named jury (the arbiter of truth). The tool feature set, test command language and emulator-based expandable test harness are described in this paper. The test harness was extended to serve as an environment that promotes higher quality products and higher productivity throughout the entire life cycle. The environment now serves as a software implementation environment, test data generator, test data manager, debugger, documentation accelerator, and regression test system.

An optimal feature set for regression testing tools for this class of embedded software product is discussed; preliminary results using the tool set are presented; and features we would like to add are listed.

Ron Blair: Mr. Blair received his bachelors degree in Computer Science from the University of California at Berkeley in 1976 and a masters degree in Computer Science from California Polytechnic State University at San Luis Obispo, California in 1977. He has worked as a systems analyst for Federal Electric, Corp., writing missile flight control software for the Air Force and in 1979 joined Tektronix, Inc., in Beaverton, Oregon, where he was a senior software engineer doing software evaluation and testing. In 1986, Mr. Blair joined Honeywell where he is currently employed as a principal research engineer. Current project involvement includes software quality and productivity improvement programs in both the U.S. and Europe.

Dan Uhrich: Mr. Uhrich received a BA in Physics from Wartburg College in 1972 and an MBA from the College of Saint Thomas in 1981. Mr. Uhrich has been a systems analyst, a software manager and a consultant. Currently he is a senior principal development engineer at Honeywell. He is the lead engineer for Honeywell's Residential Distributed Computing Systems.

Software Quality

The product discussed in this paper is the Chronotherm III thermostat made at Honeywell Inc. in Minneapolis, MN. This is a high-volume product, requiring significant cost engineering to minimize the cost of production and a higher than average need for software modifications because of regional preferences (one part of the world wants features specific to the region, e.g., language, environment and/or customs).

From a software testing perspective, the thermostat requires the smallest possible (cheapest) microprocessor capable of performing the required product functions. This means you can count on one or more code compression steps added to the development life cycle to squeeze as much functionality into program space as possible. The product should be retested after each code compression step.

Planning on multiple modifications, each with code compression steps, suggests a focus on regression testing. Without rapid retesting capabilities, the goal of product quality visibility becomes nearly unattainable. Product quality visibility refers to the ability to identify software bugs for assessing and/or improving product quality to ensure customer satisfaction.

Cost, schedule, feature set, and quality confidence are typical trade-offs. With good regression testing capabilities we have visibility of the product quality not otherwise provided. With this visibility, we are able to maintain end-to-end control of product development and make these trade-offs as sensible business decisions.

Regression Testing Requirements

Regression testing is the retesting of software. Software testing problems are universal: one must design test data, apply the data to the software under test and validate the results. Cost savings occur if test cases and test results are saved in a form that allows the test cases to be reapplied. (Changed software is executed with test cases as data or stimulus.) One may estimate the cost of testing a software product as a function of the techniques used to solve the problems described above. If retesting is required, without regression testing tools, the entire process must be repeated except for the planning. This cost is typically from 10 percent to 50 percent of the software product implementation—certainly several person-months for a nominal small-to-medium effort. Another cost savings is provided with a consistent interface, common data generation tools, and visual aids to assist in the validation of results. This implies special software tools.

To be effective the tools must remove as much of the tedium from the testing tasks as possible. Tedium causes errors and creates job dissatisfaction. Instead of pushing buttons, watching screens, moving files around the file system or other clerical chores, the engineer should be able to concentrate on test data generation methods (e.g., structured testing techniques[1]) and the application of software engineering skills. This point may appear trivial but experience suggests that if the job of testing software is not interesting, you will not get qualified people to fill the openings. Using

tools that significantly improve the test engineer's efficiency removes a previously ill understood road block and helps to provide a competitive advantage in software intensive product development.

Restricting the Problem and Refining Requirements

The scope of regression testing in general is huge. Real-time, embedded software control systems products have some features in common that allow us to narrow the scope of the problem: the systems usually have an on-board microprocessor; the software is either EPROM or ROM based; some form of physical environment feedback is detected from sensors; and the products employ various techniques to deal with signal conversions and signal processing. These techniques, implemented in hardware, have (so far) been relatively easy to emulate in software (e.g., one person-month of effort for a controller). This fact has influenced the design of the regression test tools. The generic tools such as the editor, batch test execution and report generator (see below) are interfaced to an emulator-based test harness. This approach is not always possible, especially in a larger system where it may not be feasible to create an emulator-based test harness. However, if it is at all possible, the benefits may suggest a close evaluation of the emulator-based architecture used with these tools.[2] Assuming the above common requirements for real-time, embedded software control systems, we now describe an optimal regression testing feature set.

The minimal solution requires capturing and replaying the test stimulus and response sequences, thus trapping the stimulus to and the resulting response from the software under test. The stimulus may then be reapplied to the software under test and the current response compared to the previous response. A difference in responses must mean the test or the software under test has changed. The engineer then has the job of finding out if the change is correct.

A better solution adds features such as a consistent test suite definition, database management, editing, report generation and batch test execution.[3] The features we chose for Chronotherm III regression testing include:

- A test command language,
- A test suite editor,
- The ability to totally control the test environment,
- The ability to minimize setup and shutdown time,
- Batch execution capability,
- The ability to generate reports and semi-automate the test documentation process.

These features are discussed in following subsections.

A Test Command Language (TCL)

A test command language (TCL) has been designed to provide consistency from one product to another and to give the test engineer the power of a structured language to deal with the complexities of product test. The language must be rich enough to

allow the test engineer to apply any technique desired to generate test data. A set of features is offered as a candidate for a general TCL capable of providing this richness for most software modules and more specifically for all software products for the type discussed above. Features of the TCL include:

- **SET and SHOW Commands**—These fundamental statements allow parameters to be initialized and displayed under program control.

- **BSET and BSHOW Commands**—These statements complement the SET and SHOW commands as break point set and show instructions. BSET is the desired test response and SHOW is the actual response.

- **Sequential Execution of all TCL Statements**—The current implementation interprets the TCL statements but the grammar is robust to allow compilation in the future.

- **Program Blocks**—We use the BEGIN / END convention to delineate a block.

- **Boolean Statements**—These imply arithmetic and relational expressions and operators.

- **Conditionals**—We use the block structure to augment the traditional IF statement and are not currently supporting an "ELSE" clause, for example:

  IF(boolean)
  BEGIN
  statement(s)
  END

- **Looping**—We use the WHILE statement in conjunction with the block structure to allow looping. With the features above and the while-loop, a test program may be written that conforms to structured programming concepts (one input, one output, looping and conditional execution).

- **Local Variables and an Assignment Operator**—We use the character @ to identify a local variable. Strings, real numbers and integers are supported in a context-sensitive environment. For example:

  @AVAR = SHOW TIME

  evaluates the current emulation elapsed time and assigns it to the variable AVAR.

- **Constants**—Strings, real numbers and integers are supported. Both hex and decimal integer formats are allowed.

- **ECHO Command**—The test engineer may need to send a message to the console. The ECHO command provides this function.

- **Line Continuation**—The character '\' allows lines of TCL statements to be broken in the ASCII text files.

- **PUSH, POP, RESTORE Commands**—There are times when a complex series of test statements need to be invoked to establish the proper initialization of data and control parameters. Rather than have to execute this sequence repeatedly, the total test environment may be saved with the PUSH command. POP, as expected, returns the environment to the previous state. RESTORE does the same thing except the environment stack is not affected.

- **RESET and CLEAR Commands**—The RESET and CLEAR commands allow a default initialization of the test harness parameters.

- **LOAD Command**—The LOAD command permits program controlled access to the software under test.

- **BASIC-Like REM Statement**—A BASIC-like REM statement is provided to allow internal documentation of test suites.

- **GO Command**—The GO command starts the execution of the emulators. If program termination is not assured or at least reasonable, the GO command generates an error.

- **Error Messages**—Two kinds of error messages are provided. The first is a diagnostic message indicating a syntax error or a simple early-detected problem. The other is the more serious failure such as an emulator error, memory allocation error or system failure.

- **TALK and LISTEN Commands**—These commands provide an interface with an RS-232 or RS-488 external connection. The commands are not currently implemented and are included for completeness.

- **READ and EXEC Commands**—A relatively simple macro facility is provided by allowing external files to be accessed and inserted directly into a test suite with the READ command. The EXEC command executes the file and inserts the output of the program into the test suite (not currently implemented).

- **WAIT Command**—One of the toughest problems for regression testing tools is forcing the synchronous (repeatable) execution of asynchronous events. The TCL statement addressing this problem is WAIT. WAIT has sets of arguments causing timing adjustments in the response stream from the software under test. With the current implementation, this is not a problem. If we try to execute multiple microprocessor emulators or emulate a distributed system, we may need this feature (not currently implemented).

Test Suite Editor

It is not possible to discuss editing a test suite without defining it first. In designing the suite format, we wanted to keep it simple (no exotic DBMS requirement) but we still wanted to have the ability to add and delete and navigate through a file of considerable size. The current design depends on fields of printable ASCII characters separated by delimiters that are probably never going to be found in a test suite (but we made them dynamic just in case). The lowest level entity is a test case. It is defined as containing a stimulus, a response and a documentation field. A number of test cases are linked sequentially to form a test packet. The packet also contains a documentation field. A number of packets are then linked to form the test suite. Several fields are provided to augment the documentation of the test suite in a header such as user name or date of creation. So a test suite is comprised of a linked list of packets, which are in turn comprised of a linked list of cases.

Figure 1 shows the editor window. Each of the major entities (Suite, Packet, and Case) has a name. The names are displayed in the command window. The number of the packet and case combination is also displayed. The user may navigate through the suite sequentially with the NEXT and PREV buttons or may type in a number combination and use a pop-up menu selection (right mouse button) from the CASE SELECT button to effect a "goto" function. The three SELECT buttons allow the documentation windows to pop up. The desire is to make it as easy as possible to document the test cases and packets. An engineer may choose to keep the test case small so that it can be reasonably maintained. A packet may be the collection of cases specific to a function or feature of the product being tested. At least the tool does not prohibit good documentation practices. The SELECT buttons also provide the ability to insert and delete cases and packets. Each time a new case or packet is added, the appropriate documentation window appears to allow the new test to be documented. The SUITE SELECT button allows the initialization of a new test suite file and provides a default name.

The RUN button causes the text in the middle window to be interpreted as TCL statements and executed. The results of the previous test execution appear in the lower left window. The new results appear in the lower right window. A MATCH or MISMATCH message appears in the panel above the response windows. The SAVE RESPONSE button writes the new response record over the old one. The windows can all be resized to suit the user.

Going back to the top panel of the editor window in Figure 1, the DONE button is an exit and the DISPLAY and KEYBOARD button cause pop-up windows to be created which are accelerator windows depicting the display and keyboard of the device being tested. These windows allow the user to generate TCL statements by interacting with the graphic panels and the mouse. Figure 2 shows the display panel for one product. The intent is to provide the user the ability to create segments of a test program graphically by selecting the display one would look for (break points) as a result of pushing keys (See Figure 3). The TCL statement corresponding to the icon, field, digit or item selected with the mouse is generated as editable ASCII text in the test case window.

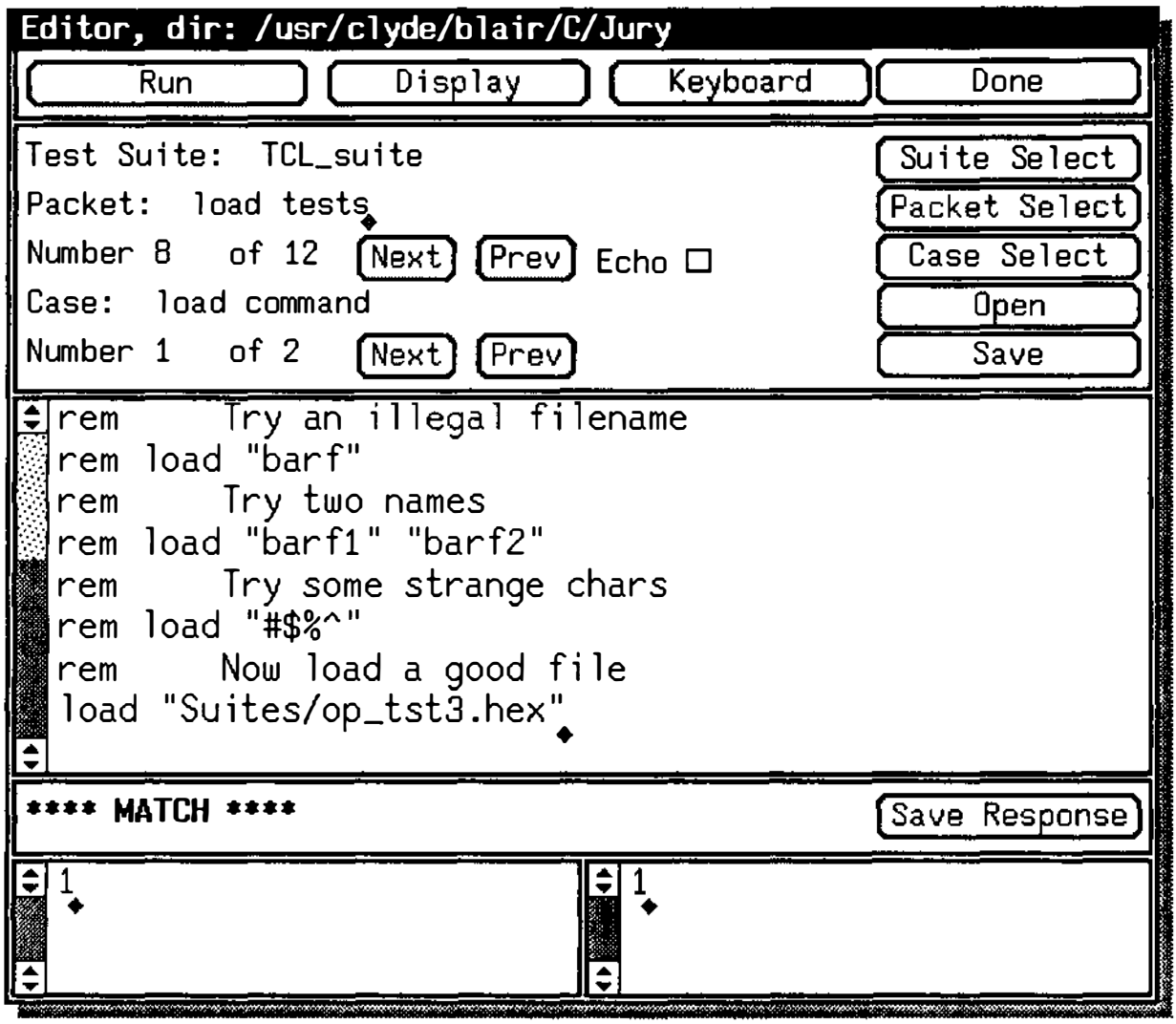


Figure 1. Test Suite Editor Window

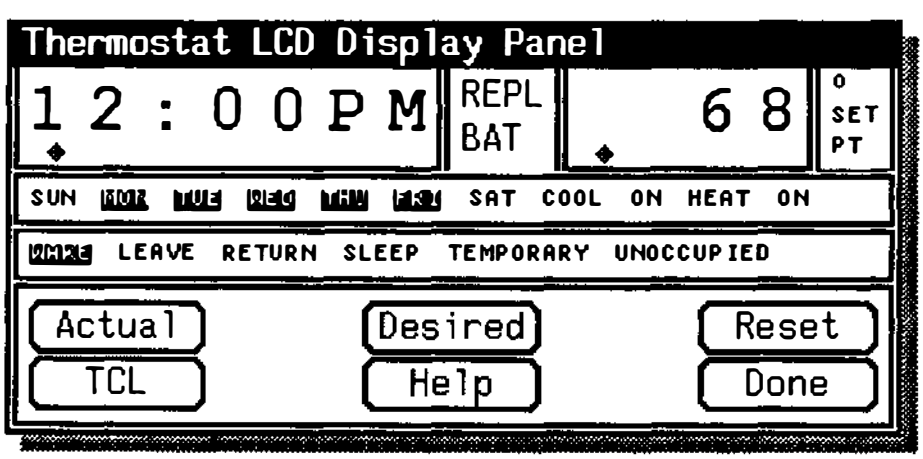


Figure 2. Product-Under-Test Display Panel

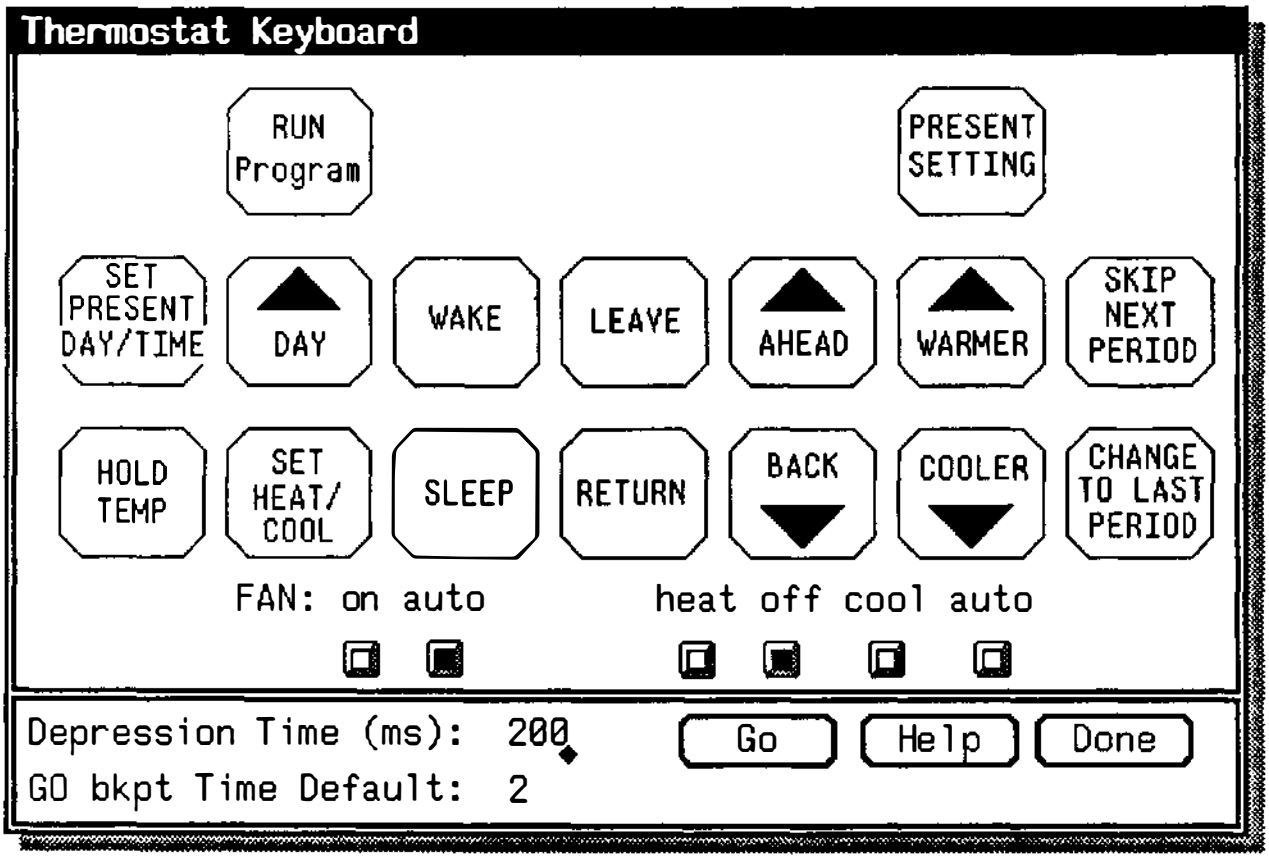


Figure 3. Product-Under-Test Keyboard Panel

As an example, consider the problem of testing the skew rate allowed for the time setting of the thermostat clock. The skew rate is the rate at which the time is displayed on the LCD display. The LCD display is updated faster if the key is kept down. The procedure is to press the SET PRESENT DAY/TIME key then hold down the AHEAD key until the desired time appears in the display. To achieve recognition requires depressing the key for a minimum of 150 milleseconds with the time advance display rate increasing thereafter according to a nonlinear curve. This is difficult to test on an actual device. Using the testing tools, the key depression time can be set with the appropriate TCL statements and an emulation break time scheduled with millisecond resolution. Using the SHOW command, the engineer may query the display (or the display panel can be observed) to measure the actual time it takes to achieve the desired setting. If the TCL supports variables and Booleans with arithmetic and relational expression evaluation, the result may be tested directly to validate it automatically. The above sequence may be accomplished using the mouse to select the product keyboard keys (Figure 3) and selecting the desired results from the product display panel (Figure 2). The graphic selections generate the TCL statements necessary to form the test case. These statements may be edited in the scrollable text subwindow for the test case.

The other buttons (Figure 1) are the OPEN and SAVE buttons. The OPEN button causes a scrollable window to pop up allowing the user to select a file or to navigate through the file system or to enter an absolute path name of the test suite to be edited. The SAVE button allows suites to be written.

All the text windows are constructed using the object-oriented SunView windowing system and inherit such features as cut and paste generally found to be productive in a development environment. We have found little with SunView to which we seriously object.

Features we would like to add to the editor include an automatic interface to a configuration control system like SCCS/RCS, test coverage analysis options and tighter integration with some of the implementation tools such as our cross compiler.

Environmental Control

Environment as used here refers to all parameters influencing the software under test but not usually thought of as necessary to control. Examples are the room temperature (the parameter presented to the thermostat emulator representing room temperature), the rate of heating or cooling and the microprocessor clock rate. These parameters must all be under strict control or the real-time feedback mechanism cannot be emulated with predictable results.

Minimizing Setup/Shutdown Time

A typical time consuming embedded software testing activity consists of stringing cables, retrieving test fixtures (sometimes competing for fixtures/devices), calibrating, etc. We have attempted to minimize this effort. The test tools are emulator based and at this time do not have any fixtures or such. All hardware elements have been duplicated in software. Some time in the future we may include purchased emulators in the test harness for new products but the software emulator-based nature of the tools should keep the loose pieces of hardware to a minimum.

Other time consuming setup costs are incurred finding files, loading files, adjusting file path names, making certain the file permissions are correct, and performing general UNIX-flavored activities. We provide a visual selection of files for test suite editing (the OPEN window in Figure 1) but depend on a good installation of the tools to minimize the clerical work. For instance, the test engineer should be logged into his/her own account; reasonable directory structure conventions should be chosen and used, etc.

Batch Execution

It is desirable to run test suites without operator intervention (e.g., over the week end). This feature is provided from the RUN window (Figure 4). A list of test suite files may be entered in the left text subwindow and execution started with the RUN button. Results of the test(s) are shown in the scrollable right window. The results include a mismatch message if the test results do not match the previous test results. This provides the user with the packet and case number where the anomaly occurred. He must then use the editor to debug the test case, edit the test case or edit the response. If the log file is needed, it may be saved with the SAVE button. Also shown in Figure 4 are various messages, which appear in the top panel. These messages indicate which packet and test case are executing during the regression test.

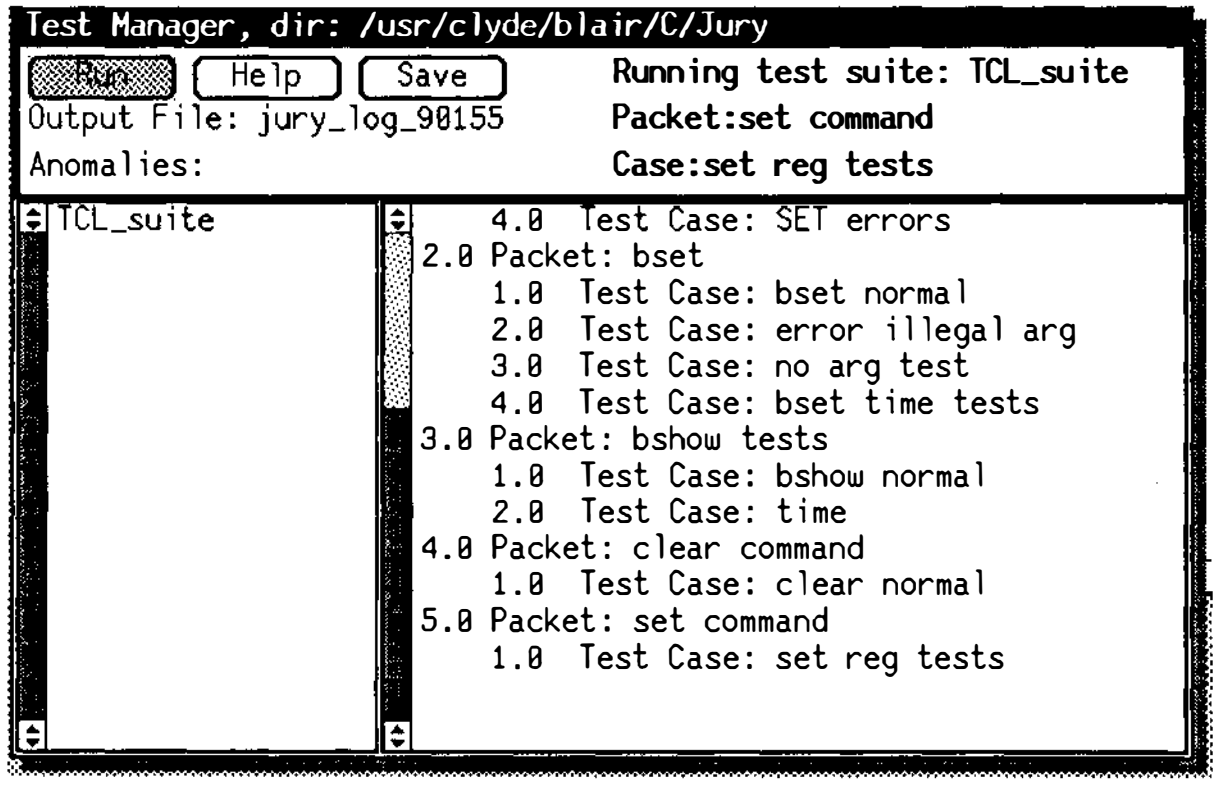


Figure 4. RUN Window

Report Generation

A large portion of the test engineer's time may be spent generating test specifications, reports, plans, etc. We have provided a REPORTS window to help in this area. The entire test suite may be printed in a "pretty print" format or the engineer may choose to generate less detailed reports. These reports are derived from the mnemonic names of cases and packets, the purpose documentation fields, and the data items automatically generated from the user's environment.

In our software development process, the software requirements specification contains a test matrix section listing the requirements with a corresponding test for verifying the requirement. The matrix should not be overly detailed and prompts the design engineer(s) to (1) think about testing each requirement, (2) offer suggestions for simplifying the test matrix (combining tests to verify more than one requirement) and (3) evaluate the cost of testing the requirement as stated. This process should be carried out as early as possible in the development cycle. The test matrix then becomes the driver for the tests and is used to build packets and test cases.

If appropriate mnemonic names are used for the suites, packets and cases, and if a concise purpose statement is included for each of these entities, reports may be extracted using the REPORTS window options which loosely conform to ANSI-IEEE-829-1983, "Software Test Documentation." The desired result is to build into the test suite as much traceability and documentation as possible. The suites are easily updated as tests are modified and we have a start to drive a traceability verification

audit. Keeping the requirement of low cost and high productivity in mind, our approach to the process optimizes product quality. The system is not without drawbacks. If more people were involved, increasing the communication requirements, the process described might not be as effective.

Interfacing the Development Environment with the Regression Testing Tools

Figure 5 shows Hewlett Packard's reproduction of a toaster model of an integrated project support environment (IPSE) framework model.[4] Most of the software environment standards can be mapped into this model. It will be some time before real frameworks like this will be available and useful for building CASE tools for processes such as regression testing. However, the model helps the tool smith better understand how to optimally partition the tool. We did not try to implement an IPSE but were able to use the object-oriented SunView interface to help map the tools into an IPSE model. With some thought to interfacing to the IPSE model, the tool becomes significantly more portable. The major IPSE components are described below.

SunView Provided IPSE Components

For our regression testing tools we used the Sun Workstation to provide the **user interface**, **task management services**, and **message server network** components in the environment (see Figure 5). This is all part of SunView. The operating system is Sun's 4.0.3 version of UNIX. The interface with C is well defined and we found no significant problems with the package.

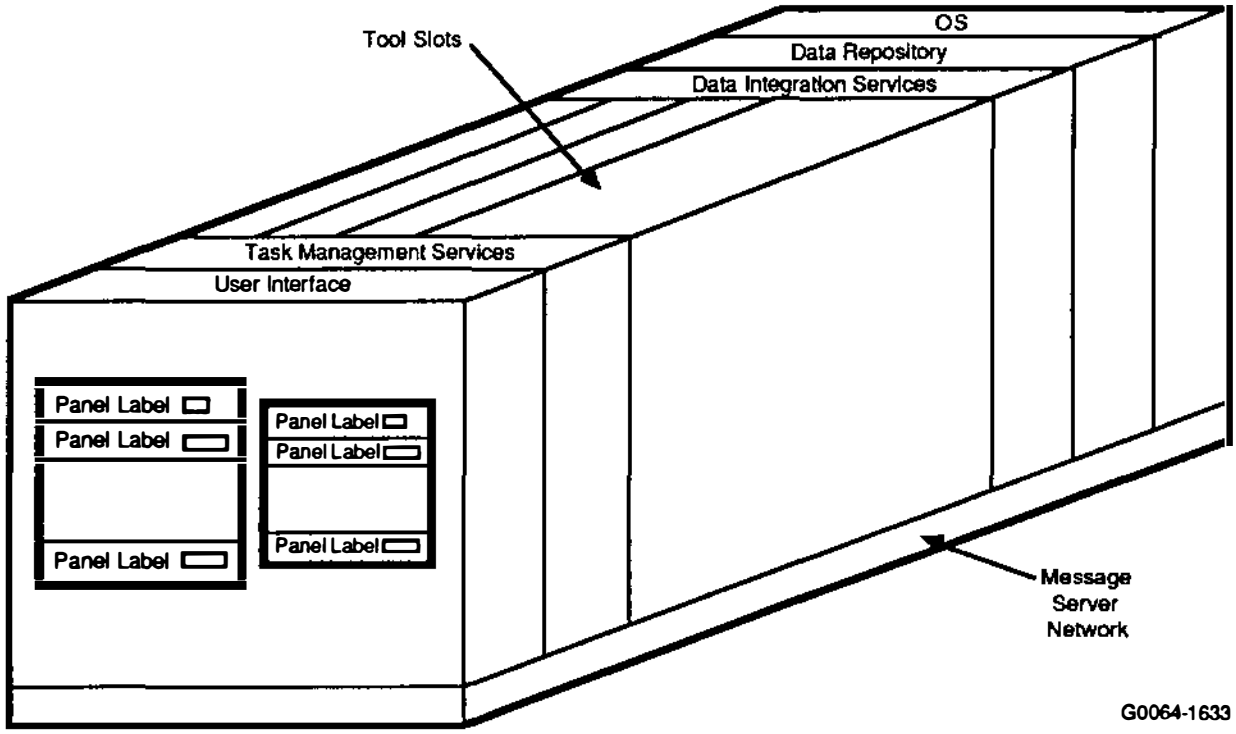


Figure 5. IPSE Framework Toaster Model[4]

Data Management IPSE Components

Since the test suite database is a simple series of records and linked lists, the data integration services and data repository services components created no surprises. It would serve no useful purpose to use an RDBMS or a more complex data manager unless the databases grow to a significantly large size or there is a need for a distributed application requirement, which is certainly possible in the future. The interfaces between these layers need to be formally defined and maintained to provide for the potential expansion of the tools.

Regression Testing Tool

The tools we are interested in are the regression testing tools and the test harness components. Figure 6 shows the main selection window of the package. The RUN, EDIT and REPORTS features are discussed above. The HELP buttons in all windows are on-line help features. The PRODUCT and MICROPROCESSOR buttons create windows to interface with the test harness. The test harness is composed of a library of product emulators and a library of microprocessor emulators each of which conceptually fit into a tools slot in the framework shown in Figure 5. From the main window the user may select the product to be tested. Each product knows what processor is required. The testing components are automatically configured for testing and may be dynamically changed under program control.

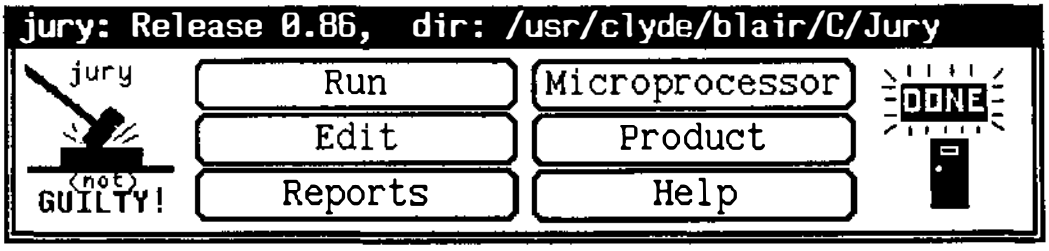


Figure 6. Main Selection Window

Product Emulator Tool

The product in this case is the Chronotherm III thermostat. The purpose of the Chronotherm III thermostat product window is to allow a user to directly set or inspect the "guts" of the product emulator.

Every key, relay, electrical component and option is displayed along with the values of various frequencies to drive the microprocessor clock and all the ambient environmental parameters such as indoor and outdoor temperatures, and battery voltage. These parameters may be set to an "actual" value or a "desired" value (break point). A default initialization file provides reasonable start up values for each product.

Microprocessor Emulator Tool

Originally we had envisioned using a simple microprocessor emulator that could run the test software and collect results to compare the next set of test results. However, there are no commercially available tools that could emulate the complexity of the NEC 7503 microprocessor used in the Chronotherm III thermostat. In addition we could not debug the product code on-line on the available tools. To resolve both issues we designed our own NEC 7503 emulator with the requisite debugging capabilities (STEP, MODIFY/DISPLAY ROM/RAM, set break points, etc.).

We used the regression testing tools to test all parts of the microprocessor emulator except the interface, which allows direct user access. The test suites for all emulators were written, edited, executed and are currently maintained by the regression testing interface. These test suites are well documented to provide on-line examples of TCL programs and tutorials of the more complex TCL features.

The microprocessor emulator window provided the basis for a complete microprocessor development environment. Software under test may be loaded directly into the emulator. The code may be executed one step at a time; the clock rate changed; and all the registers and special storage bytes changed. All the registers, I/O ports, interrupts, stack pointer, etc., are displayed; they can be observed to change as the program is stepped. In general, most of the features one could expect in a microprocessor development environment are supported or planned.

The regression testing tools form the cornerstone of a product development environment. With the ability to form tests under program control and execute them from the test suite editor, significant time is saved setting up test conditions, duplicating failure modes and isolating bugs. Additionally, these tools can test previously untestable timing-related performance features and be used to plan and execute fault-based test cases.

Architecture Summary

The microprocessor emulator is part of a planned emulator library just as the product is part of a planed product library. The regression testing tools are common to all planned emulators. The architecture is shown in Figure 7. The major features include:

- **Database Structure**—Simple–no RDBMS or such required, printable ASCII only; can be controlled with SCCS; can be mailed, etc.; fast access, easy storage , etc.
- **Documentation Accelerators**—Pop-up window support for documentation of test suite segments; automatic insertion of date, user name, etc.
- **Regression Test Tool Components**—Batch RUN, EDIT, REPORTS; capable of running infinite list of suites of arbitrary length; ease of test case access, insertion, deletion of cases or packets; three levels of reports from summary to test suite print.

- **Test Data Generator Tools**—Generate commands for test cases from graphic panels. See Figures 2 and 3.
- **Test Harness**—Microprocessor development station, product-under-test emulation, physical environmental parameter emulation, scheduler. All harness components are emulator based.

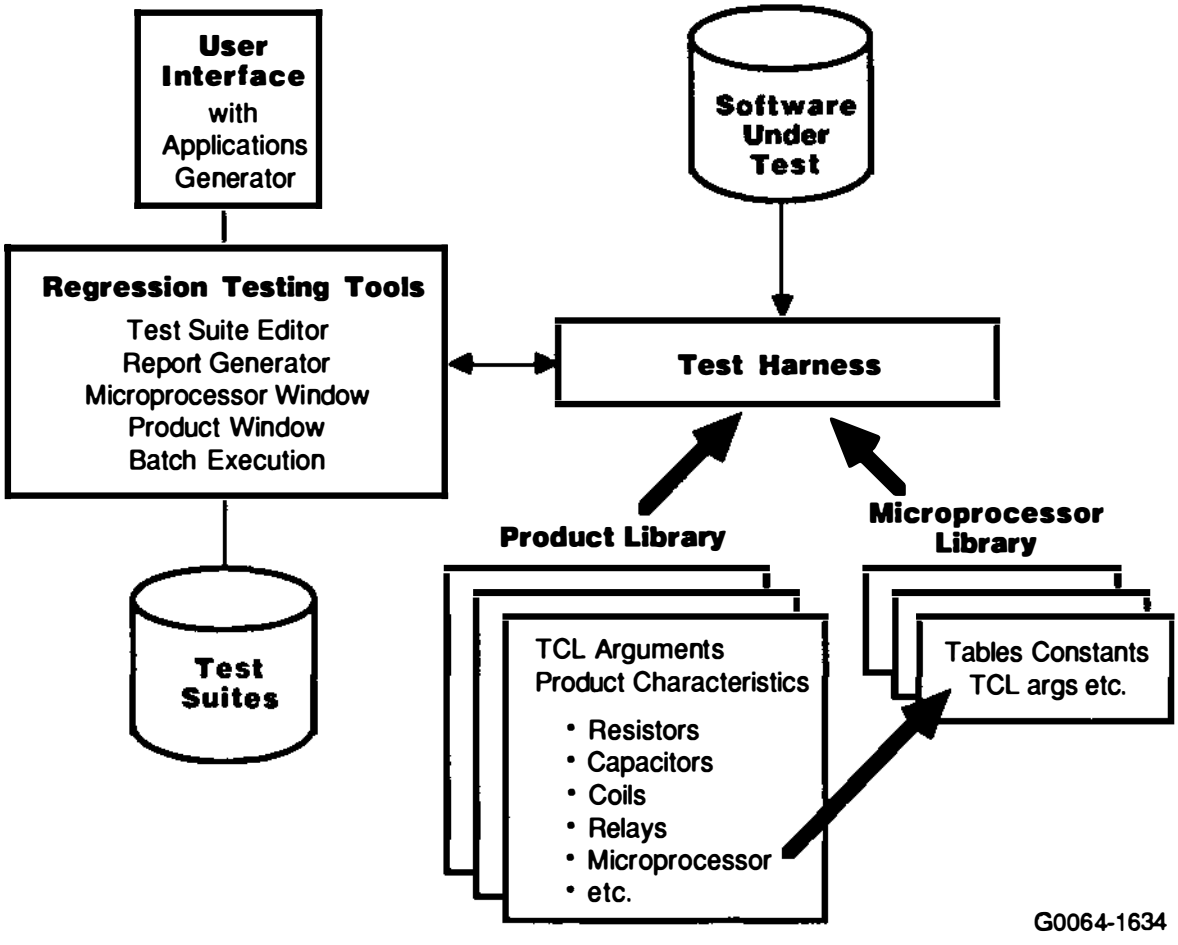


Figure 7. System Architecture Diagram

Problems

In developing our regression testing tools we encountered slow execution speed. The windowing interface does not present a performance problem nor does data management; however the execution of the emulators does. We needed to loop through the emulators every clock cycle and optimize the implementation to get reasonable execution speed. This meant getting rid of some number of floating point calculations that arguably should not have been there in the first place. The current implementation provides an emulator-to-actual product execution time ratio of ten to one. For our purposes, this is barely adequate. Further optimization is on our list of features to add.

Another significant problem for us was the "creeping feature creature." As we gained experience with the tools, we kept finding other "needed enhancements". Some of them were quick to implement but the total did create concern. We currently have a list of desired features, which continues to grow.

A problem we anticipate is one of continuing support as we are not really chartered to provide maintenance for the life cycle of the tools. One option we are investigating is to join forces with a tool vendor, provide the tool source code, and contract for the on-going support required.

Experience To Date

To appreciate the power and the utility that the tools provide, one needs to accrue some experience using the tools. There are techniques and habits that must be added or dropped. For example, when debugging assembly code it is often normal to set break points at absolute addresses. If one includes these absolute addresses in a TCL program, it works fine and provides all the debugging aid required. However, if the test case is saved to be part of a test suite, the absolute addresses are not going to be much help. If the code was not being changed, we would not be doing regression testing. If the code changes, the absolute address has likely changed also. The habit to get into is to use symbolic labels in the arguments of the TCL statements.

It is possible to set multiple break points for a test. This means that when the first desired condition is met it will cause a normal test termination. The practice may be dangerous. The result should be validated and a TCL statement provided to display the segment of the response, which is repeatable and sufficient to assure the correct result.

Because of the conventions chosen, it is possible to set a break point that is not reasonable (e.g., looking for a colder temperature while causing the ambient temperature to increase). This condition allows the program to continue without proper termination. The defaults are potentially long time periods. Therefore, one must learn to set a backup timer break point to prevent a runaway.

By establishing a set of test cases to initialize test parameters, the cases may be executed and the microprocessor emulator then invoked to allow manual stepping to verify expected results. The user may switch back and forth between the debugging environment and batch test mode to accelerate both debugging and test response validation. Again, experience with the tools is the only way to acquire techniques like this. In many instances the newly developed techniques are providing new items for our creeping feature list.

One example where the tools were expected to be of use is in the code compression step (not generally found in the software life cycle). Typically, initial values are established in the RAM and verified. A TCL program is then written to document the test. Then the code is compressed. By the time we are through, the code is not structured. The objective is to make certain that the product functions have not

been compromised. Now we load the modified code and run the test case(s). If we get a MISMATCH message, we know there is a problem. At this time we may use the microprocessor emulator to isolate the mismatch and proceed. This does not mean there are no software bugs, but it does mean that we have a higher level of confidence in the quality of the product. As bugs are found the test suite can be easily updated to include new or changed test cases stressing the detected failure.

Another example of a typical situation is a modification request for a quick change. Honeywell wants to market the product in another country where the character set has not been planned requiring different characters on the LCD display, which theoretically should be relatively easy. However, the code is compressed so that even simple changes in one part of the code effects algorithms elsewhere. Exacerbating this problem is the fact that, the code may need to go through yet another compression step. If a set of test suites exist which have extensively stressed the original product, and the mismatches found executing the suites against the modified code have been resolved, we have a much higher level of confidence in the integrity of the code.

Next Generation Tool

One of the things we want to do for the next generation tool is to provide better integration with the other development tools we use. Until the environment framework standards start to be recognized, this will be costly. There are some things we can do such as linking the microprocessor into a text editor and providing an interface to reassemble and link changed code.

The tool set described here is applicable to several types of software and the emulator-based architecture suggests that tooling costs be considered during the planning phase of a new product. Certainly the product itself must be emulated and added to the product library. If a new microprocessor is required, a new emulator must be acquired and integrated into the tool set. We know of no cheap way to do this. Using object-oriented design perspectives and the inheritance features of the method will help.

Typically, software testing activities are labor intensive. As we accrue more experience with these tools, we hope to be able to provide solid evidence that it is possible to make testing more capital intensive. Investing in tools for people, training in people skills and ways to make the job more interesting is an attractive way to improve software quality and productivity.

References

1. Perry, W., *A Structured Approach to Systems Testing: Second Edition*, QED Information Sciences, Inc., Wellesley, MA, 1988.

2. Ellis, R., "Testing of Embedded Software in Real-Time Controllers Using Emulation," *Proceedings of the Sixth International Conference on Testing Computer Software*, Washington, D.C., May 1989.

3. Leung, H., and White L., "A Study of Regression Testing," *Proceedings of the Sixth International Conference on Testing Computer Software*, Washington, D.C., May 1989.

4. Tatge, G., "IPSE Framework Toaster Model," presented as part of the National Technology University Video Panel Discussion on International CASE Architectures and Standards, January 1990.

Code Generation Using Stage on the TRANSVU II Project

Terry L. Downs
Tai-Ya Lee

AT&T Bell Laboratories
Middletown, New Jersey 07748

ABSTRACT

TRANSVU II is a new operations system developed at AT&T that will form the basis for the next generation of operations systems supporting the centralization of transport facilities administration, operations and management. The project has had significant success using Stage[1] to build code generators to generate code in a variety of areas. Stage is an application generator generator which enables the user to describe a specification language for the generator and a product description which determines the generated product based on the specification. The tool can be used to build custom generators which embody application domain specific knowledge. Stage is currently used to build code generators that generate code for the TRANSVU II Database Manager (DM), application commands, and command output functionality. The project plans to extend the use of Stage to build code generators to generate code for database masks, validation, verification, bulk loading, and mask driving. The Stage tool has increased productivity, improved quality and reduced costs for the project.

This paper suggests Stage Application Generator Generator usage concepts and presents a development model that is used in TRANSVU II. These concepts have proven to be efficient in improving the quality and productivity of the software product. The model presented in this paper explains the development process that can be implemented by a project new to Application Generator use. Two scenarios are discussed to address issues arising from initial introduction of the automatic code generation concept and the issues that surround on going use of the concept.

The use of Stage as a quick prototyping tool is explored. The approach to this implementation is explained based on the TRANSVU II experience. This approach is integrated with the two models presented to achieve improved quality and increased productivity.

This paper also presents the process that worked in TRANSVU II for the effective implementation of code generators using Stage. The process of designing and implementing a code generator using Stage is best done by the domain expert.

1. What is Stage?

Stage is an application generator generator i.e., a tool that is used to build a process which can generate code. There are two sources required to build a generator using Stage. First, a source description file is created to define a specification language. A specification, written in this language, will be interpreted by the generator to create code. Second, a product description file is constructed to define the code that will be generated based on the specification. Internally, Stage builds a parse tree of the

specification which can then be traversed to determine what code is generated. The value of tokens at various nodes in the tree can be operated on and/or substituted in the generated code. Once the generator is built, programmers then implement specifications to generate code.

2. The Command Application Code Generation Model

The bottom up approach to defining applications for Stage is used in this model. Once software module interfaces are defined for the project, applications can determine areas, referred to hereafter as components, where Stage can be used to build generators to generate code. Determination of these components requires expertise in identifying software which exhibits a small degree of variability within the component, but whose application throughout the product is wide spread. The candidate components for generation comprise code that describe a characteristic that is required in a substantial amount of features in the product. The degree of variability within the component must be small relative to the amount of total source code. This characteristic is essential because the objective of code generation is to construct a higher level language which reduces the amount of information required to generate products. If the degree of variability of a component is very high, the specification interpreted by the generator becomes very complex, requiring more information and generating less product. Figure 1, below, illustrates the structure of a feature.

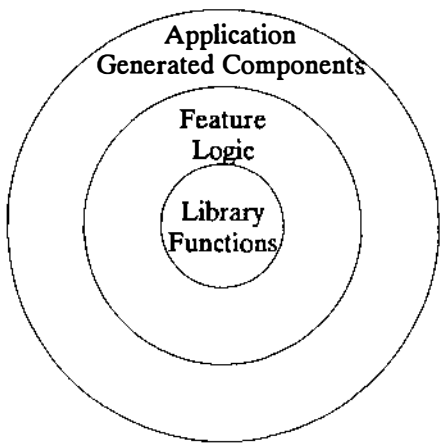


Figure 1.

Each feature is comprised of calls to library functions, feature dependent logic, and application generated components. Developers simply include the necessary application generated components in their feature. Majority of the time spent in the development effort can then be dedicated to the coding and testing of the feature logic.

The objective for using Stage is to maximize total source code produced while minimizing the specification to produce it. Once components are identified, the specification language can be developed. The generator specification describes the component and the variable parts of it. The specification language must be concise,

easy to comprehend, and extensible to achieve maximum effectiveness. The conciseness of the language makes it easy to use, maintain, and extend. It should designed using terminology that is familiar to the target development community. Extensibility of the language is important to easily allow addition of new terms while maintaining backward compatibility with previously written specifications.

The productivity gains realized using this model are determined by the number of features implemented. The earlier the application generator is developed in the life cycle of the project, the greater the productivity gain. Future generations of the product can benefit from the reuse of the generator to produce new features as the requirements are developed. Maintenance costs are reduced substantially since the application generator is maintained by a single developer. One modification to the product description can be distributed to several features requiring the change with minimal effort.

2.1 TRANSVU II Command Application Code Generation

TRANSVU II uses Stage to build generators to generate application source code, written in the C++ programming language,[2] to construct TRANSVU II commands. An application specification language has been developed, which describes the types of operations that TRANSVU II commands typically perform. The specification language is compact to reduce the quantity and the level of detailed knowledge required from developers. Some of the features offered by the specification language are: signal handling, command clean up, service opening, command line argument validation, include file declaration, and database operations, to name a few. Figure 2 illustrates the process by which the command application code generator is built and the command application code generated.

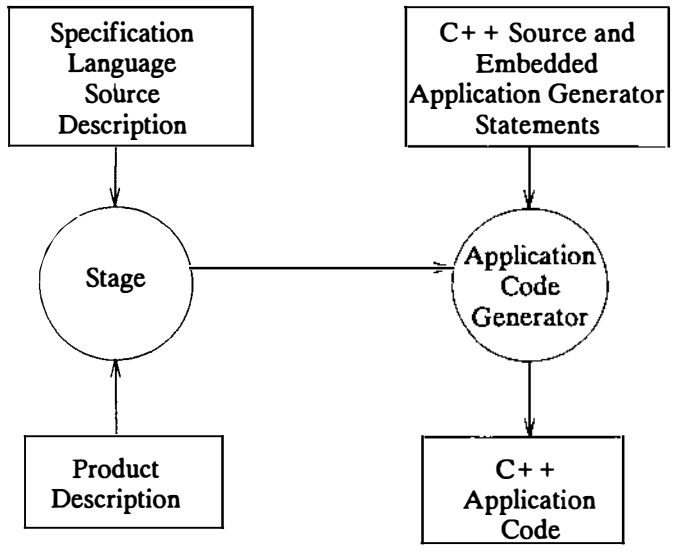


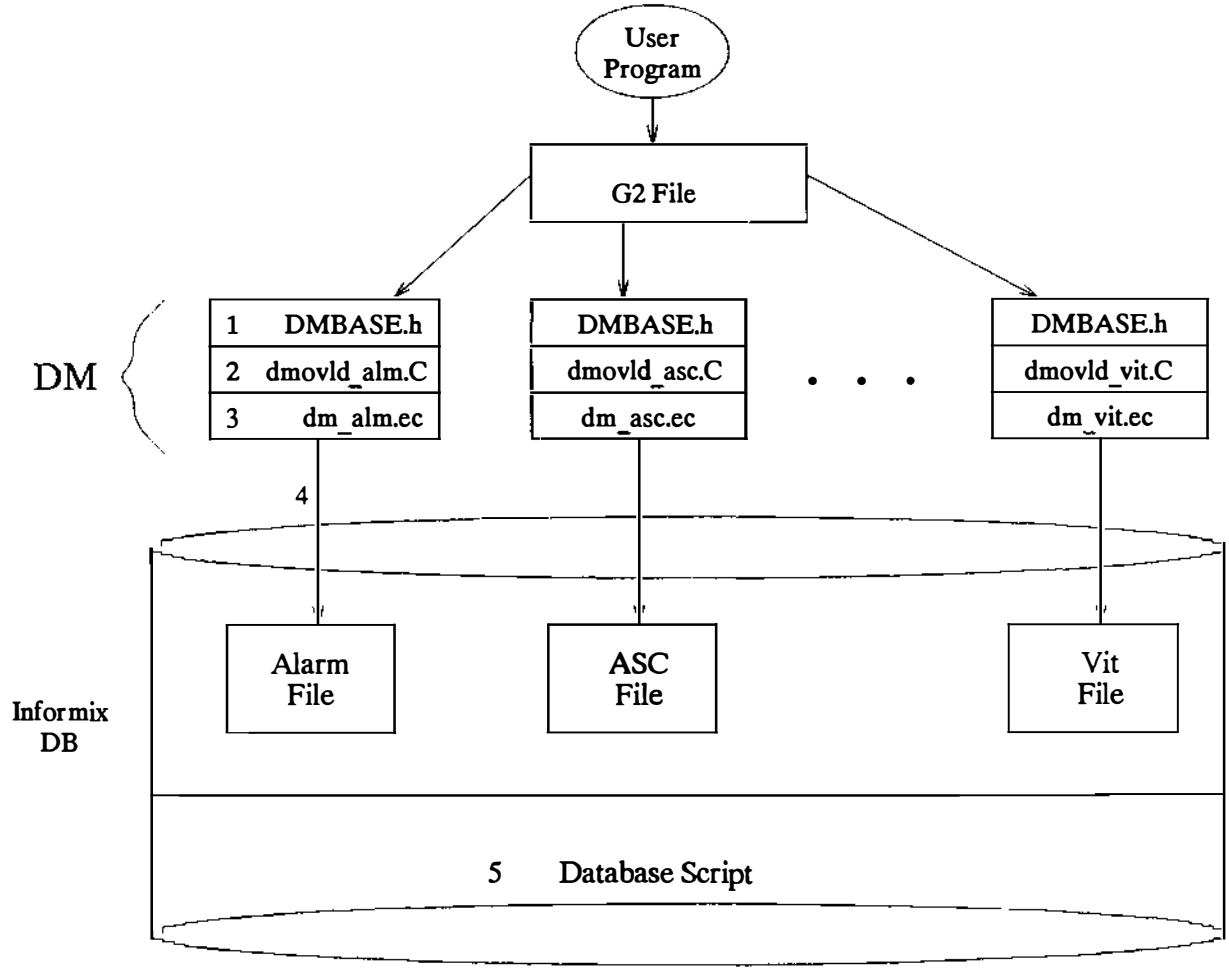
Figure 2.

The specification to the generator is C++ code with command application generator specific statements. The C++ portion of the specification is read by the generator and simply passed through in place to the product. The command application generator statements are expanded in place to their C++ equivalent. Used in this manner, the code generator acts as a preprocessor to substitute any command application generator statements in it's vocabulary with the C++ code to perform that functionality. The final product is C++ code which is then compiled to build the executable.

3. The Data Manager Model

The TRANSVU II Database Manager (DM), which is the only interface between TRANSVU II database and application programs, provides a high-level set of functions that allow application programs to access the TRANSVU II database. Figure 3 illustrates the structure of the DM software structure. It is implemented as a C++ function library which sits on top of the Informix ESQL[3]. Many of the low-level details about Informix ESQL are hidden from the user. It is a large software subsystem, and proved difficult to maintain due to the rapid change of requirements during the development cycle. Small changes to DM cause a ripple effect to occur in all other relative components of the software. Consequently, maintaining DM by providing a quick and correct fix to every requirement change was essential.

Because there does not exist a C++ preprocessor for Informix ESQL, it is impossible to factor all the common parts in every C++ class by using the base class strategy to take advantage of code reuse from an object oriented language (C++). As a result, the Informix preprocessor forces DM to keep every symbol it uses inside the file boundary which makes every C++ class very large with a high degree of similarity between them.



1. DMBASE.h defines all C++ classes.
2. dmovld_**.C defines overload functions for the class.
3. dm_**.ec file contains all DM functions and constructors for the class.
4. A 1-1 correspondence exists between DM class and physical files.
5. The Database Script can be for C-ISAM or TURBO version of Informix.

Figure 3.

Application generators are very powerful in areas where the application programs contain multiple sections of code that have a high degree of textual and semantic similarity. DM exhibits these characteristics.

3.1 TRANSVU II DM and Database Scripts Generation

Designing a specification language for an application generator may be the most important part of software generation. Theoretically, one can generate any text output from any specification, but in reality, the more the specification closely represents the logical structure of the output, the more value the user receives in terms of maintaining the output.

In the long run, generation of application programs (DM) by Stage built generators not only increases quality and productivity, but also simplifies maintenance and increases portability by maintaining meaningful specifications. The specification language used to generate DM is G2, which is a simple data language that provides a visible and program independent form for data records, representing a complete logical structure for the underlying data records. The records are sequences of ASCII characters on a byte stream and appear as indented name value pairs separated by tabs, spaces and newlines. The DM code generation model is illustrated in figure 4.

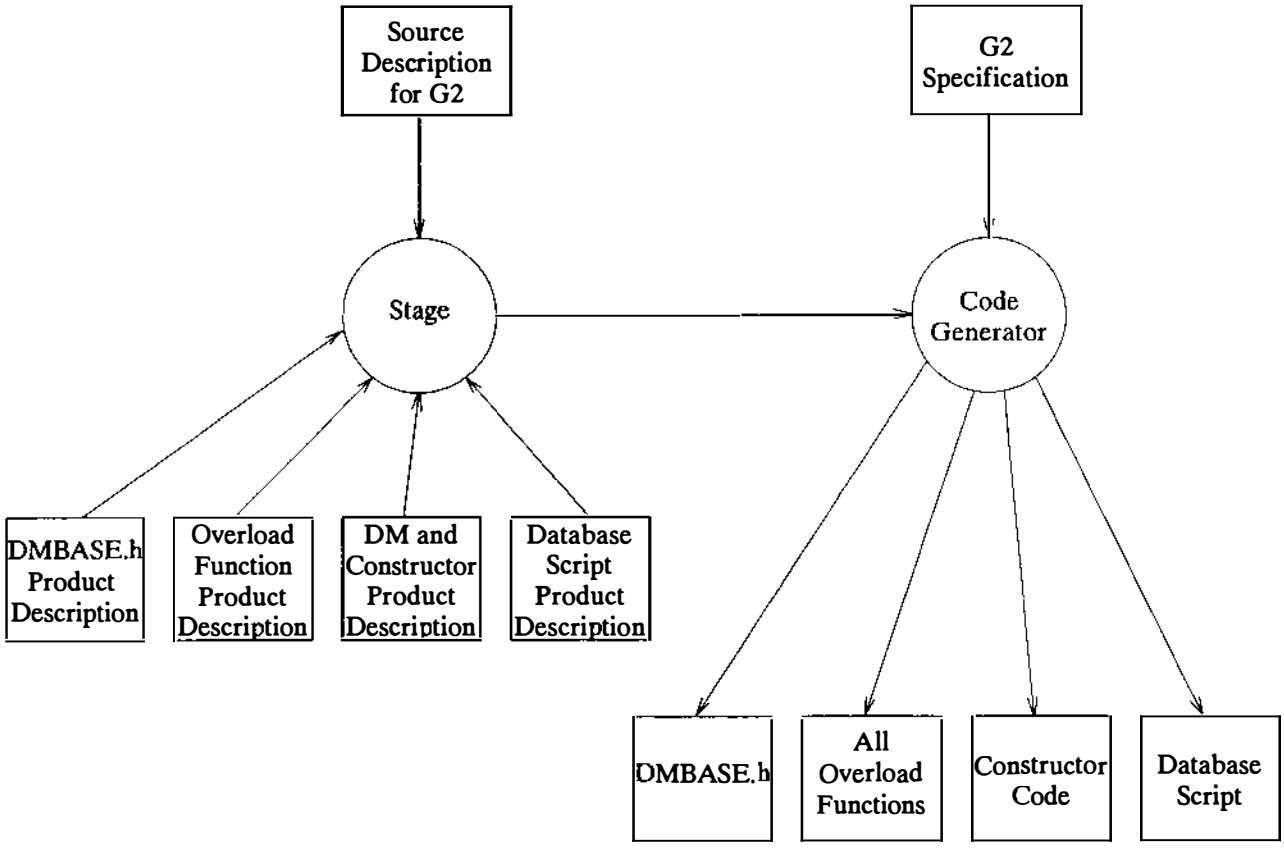


Figure 4.

In addition to generating DM using a Stage built generator, the Informix database scripts for both C-ISAM[1] and Turbo[2] database versions are also generated by the code

1. A C-ISAM database is maintained using the UNIX* file system.
2. An Informix Turbo database is maintained on raw disk partitions.

* UNIX is a registered trademark of AT&T.

** The Data Manager generator produces C++ with ESQL.

generator using the same specification. The entire TRANSVU II database including scripts and interface is a compact closed system that maintains all the components of the database and is now as simple to maintain as a G2 specification file.

The savings is overwhelming. The DM code that would take a developer two weeks to implement now takes a matter of minutes. Maintenance costs have been reduced by an equally large amount. Changing DM code simply requires changing the specification file to the generator and running the DM code generator to produce the modified code. Sixty-nine hundred lines of code are now generated which represents four percent of the total for the project.

4. Risk and Benefits Analysis

There are clearly benefits to producing code which are attractive to any software development organization large or small. Increased productivity, reduction of maintenance costs and improved quality are those the TRANSVU II project has experienced. The table below summarizes the current productivity gains for the project.

TRANSVU II Code Generation Productivity
(Non Commented Lines of Source Code)

Generator	Stage Source Code	Specification Source Code	Generated Source Code (C++)
Database Manager	1200	919	10900**
Command Output Functionality	400	1500	10300
Command Application Code	1072	200	4100

** ESQL and C++ are generated.

The productivity figures above are the most current, and it is important to note that the productivity gains will increase during the life cycle of the project. As new generics are released, the code generators are reused for new features which will increase the specification source code and generated source code columns in the table above. The source code of the generators tends to be more stable and incurs changes due to bugs found in generated code and the generation of new components of the feature. The bugs found in the generated code are found early in the process when code developers make use of the generator.

The quality gained in the code that is generated is significantly greater than the remaining portion of the code for the project for two reasons. First, the domain expert on the project is responsible for developing the code generator for that area of the

project. Second, the code that is generated is less complex in most cases than other parts of the project coded by individual developers. Below is a summarization of the modification request types currently entered against the application code generators discussed in this paper.

TRANSVU II Code Generation Quality
Modification Request Types For Code Generators

Generator Functionality	TRANSVU II Generic	C++ Compiler Generic	Generated Code Faults	Total MRs
24	5	6	18	53

To measure the quality of the code generated the total lines code of code generated (15900) is divided by the number of code faults identified (18). Hence, the code fault density for the generated code is 1.13 faults per 1000 lines of code generated. The reuse of generators to generate code for new generics of the product will reduce the fault density.

Substantial gain is realized from code generation by the standardization of code. The code which is generated is standardized across the project to meet requirements for the system.

Risk and benefit analysis must be performed up front prior to designing and implementing an application generator to insure that the cost of building the code generator is less than the total cost of building the code it generates. Once this determination is made, the aforementioned benefits will follow.

The success of the TRANSVU II project is due, to a large extent, to the use of Stage as a tool to increase quality and productivity. Software has been produced ahead of schedule and with greater quality. The re-use of generators in successive generics has even further justified their cost to build.

REFERENCES

1. *AT&T Technical Journal - Software Productivity*, "Tools For Building Application Generators", J. Craig Cleaveland, Chandra M. R. Kintala, July/August 1988, Vol. 67, Number 4.

2. *The C+ + Programming Language*, Stroustrup, B., Addison-Wesley Publishing Co., 1986.

3. *INFORMIX - ESQL/C Relational Database Management System Reference Manual,* Informix Software, Inc., July 1987, Version 2.10.

Seams and Cloth, the Future of Computer Science

Dr. Mark Weiser

Xerox PARC
Computer Science Laboratory
333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

Most reseach in computer science is "whole-cloth", in which one constructs a complete computer system, without much regard for its interoperability with other software. Examples of such whole-cloth research are Unix and C + +. In another view, interconnection of systems is the prime focus. Examples are program slicing and Sun's NeFS proposal. In "seam" research the interconnection of systems has as much challenge as inventing "whole cloth." As the computer world becomes increasingly heterogeneous, seams research will come to dominate computer science.

Biographical Sketch

Dr. Mark Weiser is Principle Scientist and head of the Computer Science Laboratory at Xerox PARC. Dr. Weiser has helped start two companies and been a tenured professor at the University of Maryland. He likes to blend philosophy, psychology, hardware and software in his work. His most recent scientific work is on new methods of automatic storage reclamation. He has been working on the next revolution of computing after work-stations, called "Ubiquitous computing".

Seams and Cloth: The Future of Computer Science Research

Mark Weiser

Xerox PARC

Outline

Computing Today, Seams vs. Cloth

Why seams are so important

Seams of the Past

Seam Work Today

Seam Work of the Future

Alternate Titles

"my path through the research jungle"

"a philosophy of systems"

"hermeneutic computer science"

"a (less) humble programmer"

Computing Today

- **like medieval alchemy**
  - has some success
  - revered
  - does not understand *why* things work
- **like medieval astronomy**
  - very important technology
  - navigated to new world using incorrect astronomy
  - used compass, no idea how or why it worked
  - was monolithic, seamless, had to be taken as a whole cloth

What computing today is good at

- **MIPS (e.g. fast processors)**
- **Single purpose systems**
  - airline reservations
  - shuttle control
  - single languages (C, Ada, Prolog)
  - single systems (PC's)
- **These are all cloth**

What computing today is bad at

- **connecting machines together**
  - email (except monolothic email)
  - LAN's (except for monolithic LANs)
- **multipurpose systems**
  - cooperating PC software (MS Windows)
  - workstations
  - non-monolithic integrated environments
- **These are all seams**

Summary so far

We understand our cloth, our 'things' but not how to join them into clothes.

We don't understand the *relations* among our things

Seamless systems is a false goal – better are *seamful* systems in which we understand the seams

Understanding seams means *real* understanding

Physics Analogy

Early physics concentrated on elements and their properties: earth, air, fire and water

Advances in physics have, very broadly, all been changes from understanding elements to understanding relations

- gravity is a relationship between masses
- Maxwell's equations, fields, are about relationships, not objects
- relativity
- particle physics now defines particles in terms of their interactions

Computer science is still mostly at the elemental stage, looking at things and properties, not relations.

elements = cloth

relations = seams

Examples of cloth in computing

Unix

Algorithms

Any one communications network

Any one compute r language

These are each powerful in themselves, but are a limited, focused, closed world.

Examples of seams in computing

Interoperability

Linguistics

Situated Use Theory

(new) Operating system kernels

Meta-documents

Collaborative tools

Abstract Datatypes

Object orientation

These each are about the connections among other things assumed to be pre-existing in the world

Why are seams so important?

Computer science is intoxicating.

We computer scientists pour ourselves into systems we completely build, control, understand, calculate.

We fool ourselves, but "nature cannot be fooled", and nature does not respect our system boundaries.

For long-range success we *must* understand how to design, build, and model systems that we do not control – systems that touch the world.

Only when our work is thus in contact with nature are we scientists, not tinkerers.

Examples of seam research

PAST: program slicing

PRESENT: PARC projects on meta-documents and psychophysical documents

FUTURE: Interoperability Theory; Loophole Science; Bigger, broader human/computer seams (future PARC projects?)

past seam research
Program Slicing

- **Given:**
  P, a program
  v, a variable
  s, a statement
  $P|_{(v,s)}$ denotes a slice, defined to be exactly those statements in P necessary to produce all behaviors of v at s
- **Example**
  s = 9, v = "a"

	P:	$P|_{(v,s)}$
1	a := 0	★
2	read(x)	★
3	y := x + 1	★
4	z := x + 2	–
5	if (y) then	★
6	a := y	★
7	b := z	-
8	end	
9	write(a)	★

Interesting things about slices

- **Slices of programs are themselves programs, a union of slices is a slice, ...**
- **Slices are mathematically interesting:**
  growing body of literature
- **People think this way**
  several experiments show slices correspond to a natural and powerful view of programs
- **Slices are a kind of calculus for developing, modifying, parallelizing, debugging programs**
- **Slices are orthogonal to every conventional program structuring method**

Slices as Seams

- **Seam between observer and program**
- **Seam between program behaviors**
- **Seam between program changes**

present: Meta-Documents

Example Goal: "electronic documents as easy to use as paper"

Key Problem: document interchange

"cloth" mentality solution: standardize on a single document representation

"seam" solutions:
meta-documents
system 33

MetaDocuments

"mini-languages" - Al Aho, AT&T Bell Labs

- domain specific
- encapsulate expert knowledge and algorithms

Examples:

Postscript, Unix "grep"

People come in contact with documents via their eyes and editors, so the mini-language needs to be about the editability and appearance of documents

System 33 documents as psycho-physical effects ("appearance")

more than we edit, we read

paper documents require only a good light and a decent chair

electronic documents require display device, proper resolution, format conversion, preprocessing, processing, and postprocessing, ...

SOLUTION: seams: between person and document, between workstation and server, between document and database

A document *is* a psychophysical effect on retina and brain

System 33

a system for storing and retrieving psycho-physical effects

call psychophysical effects "appearances"

abstract appearances have a name, called a "handle"

abstract appearances have descriptions which are simple ascii.

Using System 33

query into appearance descriptions yields handle

handle plus rendering spec yields array of light and dark

rendering spec via many different forms: Postscript, cloudy football stadium screen, IBM PC display in sunny office

System 33 promise: to render as best it can the Platonic appearance to your eye via your rendering device

System 33
How It Works

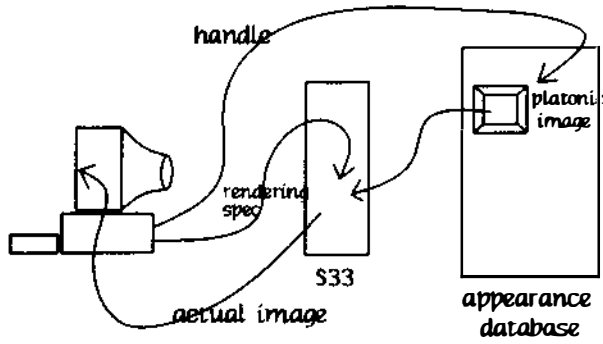
A. System 33 is an expert at format conversions

B. Sophisticated scaling and bit tuning for rendering devices

C. Protocol among computers using System 33 permits wide variation in rendering speed, caching of results, offloading work to either client or server.

D. Complicated work done by System 33 so document interchange is trivial among machines (i.e. a kind of electronic super-Fax)

System 33 picture



Advantages of Appearance Approach

very simple clients

universal exchange

constant improvement

from seam thinking, not cloth

System 33
Current Status

work began January 1988

First prototype running demonstrations in April 1988.

Now many prototypes in use inside and outside Xerox.

User interfaces for Suns, PC's, Macintosh's, Lisp, Cedar, Smalltalk, Viewpoint, Fax,...

Larry Masinter now in charge

Working now on reliability, performance, additional capabilities

Future: Seam Science

Specific for instances:
- **Interoperability Theory**
- **Meta-Protocols**
- **Bigger, broader human/computer seams**

Interoperability Theory

Q: what does it mean for a program or file or machine to interoperate with another program or file or machine?

A: nobody knows, but everyone knows it when they see it

Examples: exchange spreadsheet data, sending email,...

Seeds of a Theory of Interoperatibility

A interoperates with B for job J if A can get B to do J.

carried out in automata theory, for universal interoperability classes

A universally interoperates with B if, for all J that B can do, A can get B to do J for it.

class *A* universally interoperates class *B* if there exists an A in *A* such that for all B in *B*, A universally interoperates B.

Class FSA does not universally interoperate itself, class LBA does, PDA universally interoperates FSA, ...

not really yet capturing interoperating – but may be on the way

Loophole Science

Two ways to design systems:
- **old: specify functions**
- **new: specify freedoms (i.e. loopholes)**

Examples:

telephone system since Carterphone

Phase III ("System Vr5") Unix is apparently designing in terms of extendability, freedom, loopholes

The Mach operating system is not so new in its functions, but IS new in keeping them small, and making them available for exploitation

PARC's Computer Science Lab's new multilingual environment opens itself by replacing constants with procedures, thereby multiply/simultaneously customizing itself for many languages

Object-Oriented-Programming's power is from replacing function with freedom (e.g. "Here is a mesage, do whatever you want.")

Common Lisp Object System

(describe the meta-protocol here, and why more generality also leads to more performance)

Bigger Broader human/computer seams

Computer interaction now is awful
not just because of crude applications
not just because of bad user interfaces
not just hardware limitations

mostly because of complete lack of fit with everyday human activity

Using a computer is like going to the Dentist
bright lights, noises, useful things getting done, but sterile environment, you go to it (even for laptops!), awkward, painful

Little pieces of glass, and lots of little buttons, are absurd.

What's wrong with seams?

- **Vague, unscientific**

  but all science starts with intuition, a "feeling for the organism"

- **inhibits creativity, unnecessary constraints**

  but the world, not I, force seams
  you can only ignore some of the world some of the time
  seamless research also necessary, but necessarily short-lived

- **solved by standards**

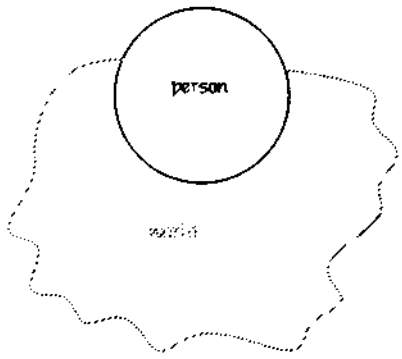
but yes, in some cases
but we cannot standardize the universe, nor would we want to
true computer science laws will find unity in diversity, not eliminate diversity

A coloring book

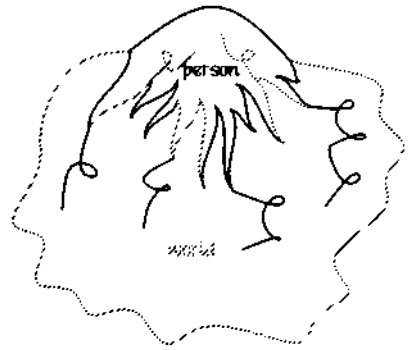
on

hermeneutic phenomenological system design

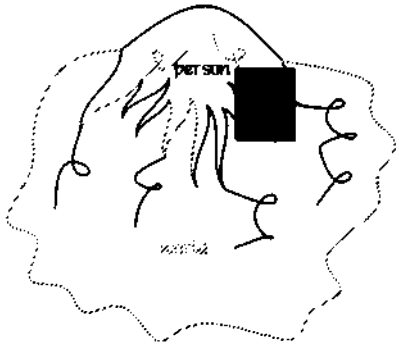
Wrong



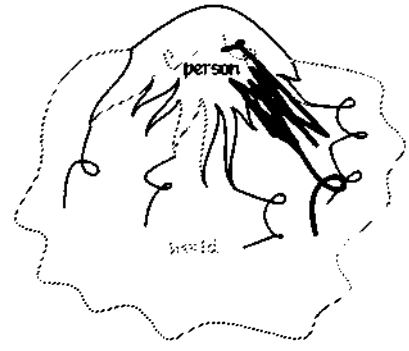
Right



Cloth Technology



Seamful technology



Any work can be seamed

minimize control

maximize co-existence

simplify interfaces

open loopholes

broaden assumptions

use "situated thinking": little prior knowledge, much local knowledge, the world as given

Summary

Computing is still at "earth, air, fire, and water"

Relationships, seams, will rightfully come to dominate over elements in our theory and practice

Completely new science will emerge from understanding relations

Not seamless systems, but beautiful seams

The future is in the seams

A Slicing/Dicing-Based Debugger for C

Sekaran Nanja and Mansur Samadzadeh
Oklahoma State University
Computer Science Department
219 Math Sciences Building
Stillwater, OK 74078
E-mail : samad@a.cs.okstate.edu

Abstract

Debugging or the isolation and removal of errors encountered in unfamiliar programs (i.e., programs developed by someone else), is a difficult and time-consuming task. Ideally, debugging requires a thorough understanding of the program under consideration. This can be accomplished by following methods such as reading and comprehending the whole program, testing the program with a random set of inputs, tracing the program modules dynamically, reading the program documentation, or a combination of these methods. Each method requires the programmer to conduct an extensive and detailed analysis of the program in order to isolate the reason for the occurrence of even a relatively insignificant error.

Minimization of this excess information is accomplished by program slicing, which is a technique based on data flow and control flow analyses. Program slicing decomposes a program into relatively small portions called slices. The resulting slices contain the statements which are relevant to the computation of particular outputs. Dicing, an application of program slicing, is the process of identifying a set of statements likely to contain the fault(s) causing a given output variable to have an incorrect value. This paper briefly describes program slicing and dicing with a number of examples, provides details about C-Sdicer (a slicing and dicing based debugging tool for C), and explains a prototype evaluation of C-Sdicer.

Biographical Sketch:

Mr. Sekaran Nanja received the Bachelor of Engineering degree in Mechanical Engineering from University of Madras, Madras, India, in 1983, and the Master of Science degree in Computer Science from Oklahoma State University, Stillwater, OK, in July, 1990.

Dr. Mansur Samadzadeh received the Bachelor of Science degree in Computer Science from Sharif University of Technology, Tehran, Iran, in 1978, and the Master of Science and Ph.D. degrees in Computer Science in 1982 and 1987 from the University of Southwestern Louisiana, Lafayette, La. He is currently an Assistant Professor of Computer Science at the Computer Science Department of Oklahoma State University. His research area is software engineering.

Program Slicing

Program slicing [Weiser 81 and 84] is a method of decomposing a large program into small portions based on slicing criteria. A slicing criterion is a tuple <n,V> where n is the source program line number and V is a set of variables currently under investigation. The resulting slice is a set of statements which influence V from the line number n of the program back to the beginning of the program.

A program slice can be considered as a projection of the original program [Weiser 84] that produces the same result as the original program for the variables in V starting from the beginning of the program up to line n of the original program. Data flow and control flow analysis techniques [Hecht 77 and Barth 78] are used to find the statements which influence the criterion set of variables V.

Example of Slicing: Figure 1(a) is a program which reads two numeric values and then calculates and prints the sum, the average, and the maximum of the input values. Figure 1(b) depicts the program slices for the criteria <8,{sum}>, <8,{max}>, and <8,{a,b}>.

```
1    readln(a,b);
2    sum := a + b;
3    avg := (a + b)/2;
4    if (a > b) then
5         max := a
6    else
7         max := b;
8    write(a,b,sum,max);
```

(a) A sample program segment.

```
Slice on criterion <8,{sum}>
     1    readln(a,b);
     2    sum := a + b;

Slice on criterion <8,{max}>
     1    readln(a,b);
     4    if (a > b) then
     5         max := a
     6    else
     7         max := b;

Slice on criterion <8,{a,b}>
     1    readln(a,b)
```

(b) Examples of slices.

Figure 1. A sample program and some of its slices.

Program Dicing

Program dicing [Lyle 84 and Weiser 86] is the process of identifying a set of statements likely to contain a fault. Dicing is an application of program slicing.

In a typical debugging session involving program slicing, a programmer slices the program in question based upon incorrect output values(s). The programmer then narrows down his/her field of vision to comprise only statements considered relevant in identifying the possible cause(s) for the incorrect output value(s). However, in the case of program dicing, both correct and incorrect output values are used. This is done with the objective of eliminating the statements that appear to be correct from the slice that was generated based upon incorrect output values(s). A dice consists of statements in the slice generated with respect to incorrectly valued variable(s) while excluding any statements in the slice pertaining to variables(s) that appear to have been computed correctly. From the foregoing discussion, it is clear that dicing produces fewer statements than does slicing. The only exception that could occur is when slices of correct and incorrect variable(s) have no statements in common. Such a situation could aptly be described as being the worst case possibility that could arise [Lyle 84].

Example of Dicing: The following example [Lyle 84] should help in clarifying the role played by dicing in the process of fault localization.

```
1    readln(x,y);
2    sq_x := x*x;
3    sq_y := y*y;
4    prod := - x*y;
5    sq_p := prod*prod;
6    write(sq_x,sq_y,prod);
```

Figure 2. A program to print squares.

In this example the intended value of variable "prod" should be equal to x*y and not -x*y. This fault exists in statement number 4. Testing the output variables at statement number 6 will generate correct output for all variables except for the variable "prod". The slice on variable "prod" at statement number 6, <6 ,{prod}>, is {1,4,6} because statement number 6 is the statement under consideration, the value of "prod" was last changed in statement number 4, and the values read in at statement number 1 are used in the right hand side of the assignment statement for "prod". The slice on "sq_x" at statement number 6, <6,{sq_x}>, is {1,2,6}. Hence statements number 1 and number 6 in the slice are correct based on the slice on "sq_x". Eliminating these two statements from the slice on "prod" will result in statement number 4, the faulty statement.

C-SDICER

Based on a review of the literature, there seems to be no slicing-based debugging tools currently available for the C programming language [Kernighan 78]. A slicing/dicing-based debugging tool called C-Sdicer (C-Slicer/dicer) was developed. The rest of this section is devoted to a detailed discussion of C-Sdicer and some related implementation issues.

Design Approach: This section describes the design approach adopted in the development of C-Sdicer. C-Sdicer provides an effective debugging environment. It offers various commands such as slice, dice, edit, run, and compile.

The slicing portion of C-Sdicer relies upon an adaptation of Lyle's algorithm [Lyle 84]. Lyle's algorithm computes an active set for each statement in a program. These active sets are stored in memory and are instrumental in the computation of slices and dices. The algorithm employed for C-Sdicer computes active sets dynamically. A reduction in space complexity is achieved with the dynamic computation of active sets. However, this advantage of reduction in space complexity requires that a price be paid for it. The price to be paid is exacted in terms of a marginal increase in computational time complexity. The slicing algorithm employed for C-Sdicer initially computes slices, consisting of statements that affect the slicing criterion directly, followed by those statements that affect slicing criterion indirectly. This process is carried out recursively.

C-Sdicer computes slices of C programs that are devoid of function calls and expressions involving pointers and bitwise operators. C-Sdicer is mainly aimed at novice programmers and is expected to be of value to beginners. The dicing algorithm used in C-Sdicer corresponds to one presented by Lyle [Lyle 84]. This dicing algorithm computes slices based upon both correct and incorrect variables. Subsequently, the algorithm eliminates statements common to the two sets of slices from the slice generated with respect to incorrect variables only, thus generating a dice.

C-Sdicer User Guide

This section describes the use of C-Sdicer in the debugging of C programs.

The C-Sdicer environment permits the use of a restricted set of commands. C-Sdicer also permits the execution of any shell command within its environment. However, it is necessary to precede the chosen command with an "!".

The C-Sdicer command set is presented in the following paragraphs.

s or **slice** The arguments for this command are a line number and a set of variables separated by spaces. If proper arguments are supplied, this command produces a slice.

d or **dice** The arguments for this command are a line number, a set of variables separated by spaces (the incorrect variables), a partition marker (|), and another set of variables (the correct variables). This command produces a dice.

cat or **type** This command displays the program on the video screen.

l or **load** This command loads a new file into the C-Sdicer environment. **Load** accepts a filename as its argument.

e or **edit** This command invokes the **"vi"** editor and allows a user to edit the program currently resident in the working environment.

c or **compile** This command invokes the C compiler with the program currently resident within the C-Sdicer environment as its argument.

r or **run** This command executes the program currently resident within the C-Sdicer environment.

man This command provides on-line help for the entire C-Sdicer command set. The **man** command accepts a valid C-Sdicer command label as its argument. Upon execution, syntax and usage directives for the user supplied command are displayed, accompanied by an illustrative example.

Debugging Using the C-Sdicer Approach

This section describes the utility of C-Sdicer in debugging C programs.

C-Sdicer generates slices and dices based upon specific, user-supplied criteria. A slice generated thereby consists of all statements relevant to the slicing criterion variables specified by the user. The slice spans the length of the program up to and including the statement in the slicing criterion. On the other hand, dicing, an application of program slicing, generates relatively fewer statements which are the ones that are likely to contain the error(s).

One way of observing errors in a program is through output statements (write statements). To find the statements which are responsible for the erroneous output value(s), the program in question is sliced at that particular write statement, with the erroneous variable(s) as parameter(s). However, some of the variables, contained in the write statement being considered, could possibly be identified as being obviously correct. In such a case,

these variables could be used to dice the slice based upon the incorrect output variables.

In short, C-Sdicer provides an environment which allows its user to edit, compile, and run the program currently resident in it. Thus, C-Sdicer provides a complete environment that could be used to debug C programs.

Advantages and Limitations of C-Sdicer

This section highlights the advantages and limitations of the current implementation of C-Sdicer.

Advantages

1) C-Sdicer generates slices based upon a set of user-specified variables at any particular statement in a program.
2) C-Sdicer generates dices with respect to a specific dicing criterion.
3) Several options such as edit, compile, and run, which are found to be useful in the process of debugging, are available in the C-Sdicer environment.

Limitations

1) The current implementation of C-Sdicer is intended for use by novice programmers. Consequently, it is not equipped to handle programs containing expressions involving pointers and bitwise operators.
2) C-Sdicer currently cannot handle programs with function calls.
3) C-Sdicer is designed to handle programs which are devoid of syntactic errors.

Preliminary Evaluation of C-Sdicer

One way of evaluating software is by conducting a controlled experiment [Conte 86]. In an experiment that was conducted as a preliminary evaluation of C-Sdicer, a group of subjects were asked to perform tasks using C-Sdicer.

There are in general three phases in a controlled experiment. The first phase, pre-test, involves the collection of relevant information concerning a potential subject group for initial screening. In the second phase, treatment, any information that may be required, as regards the piece of software that is being evaluated, is disseminated to the group of subjects that have been selected. The last phase, post-test, involves the actual usage of the software being evaluated by the subject group and the collection of the relevant information.

C-Sdicer was prototypically evaluated for its effectiveness and utility as a debugging tool. The study involved the participation of a group of computer science students at Oklahoma State University. The following sections describe the subjects and

tasks involved, and the procedures employed in the evaluation of C-Sdicer.

Subjects. The students involved in the evaluation process were enrolled in the following computer science courses taught at Oklahoma State University.

1. Operating Systems I
2. C programming language
3. Operating Systems II

Of these courses, Operating Systems II is a graduate course. Hence, the subjects were composed of both graduateand undergraduate students. This mixture of both graduate and undergraduate students was considered desirable in order to ensure the participation of both expert and novice programmers in the study.

Tasks. The student participants involved in the evaluation of C-Sdicer were asked to make use of their own C programs which were devoid of function calls (except for C built-in functions) and expressions involving pointers and bitwise operators.

Evaluation Procedure

The process of evaluation of C-Sdicer involved three major steps. These steps are discussed in this section. It should be noted here that the pre-test phase of this experiment was implicit, in that enrollment in a certain undergraduate or graduate course was considered as having passed the initial screening stage or the pre-test.

Step 1 (Treatment). In this step, a brief presentation encompassing program slicing, program dicing, use of C-Sdicer in debugging C programs, and a discussion of the capabilities of C-Sdicer was made to the subject groups. An informal question/answer session was also conducted to enhance the understanding of the participants in the proper use of C-Sdicer. A two-page handout consisting of all of the above-mentioned details was distributed to all members of the subject groups participating in the evaluation process. The on-line help inside the C-Sdicer environment, which provides details about the usage of various commands and their syntax, is considered part of the treatment also.

Step 2 (Task). In this step, the members of the subject groups were asked to make use of C-Sdicer while developing C programs to view slices and dices based upon various criteria to debug their programs. However, only C programs that satisfied the constraints mentioned in the section on tasks were to be used in the evaluation process.

Step 3 (Observation or Post-Test). This step was crucial in the evaluation of C-Sdicer. When a subject exited the C-Sdicer

environment, he/she was requested to answer a series of simple questions on a voluntary basis. If a subject chose to answer the questions, his/her responses were appended to a file. If a subject elected not to answer the questions, an 'N' character was appended to the file.

Results and Conclusions

The following are the results and conclusions based on the responses tendered by the subject groups.

1. In all, one hundred and forty subjects (students) were asked to use C-Sdicer. However, only forty two subjects made use of it.
2. Of the forty two participants, twenty nine subjects chose not to answer the questions.
3. Most of the subjects involved suggested that C-Sdicer should be enhanced to handle programs containing function calls and expressions involving pointers and bitwise operators.
4. Based on the responses submitted, we can informally arrive at the conclusions that follow:
   a) Most of the subjects involved felt that C-Sdicer's help features were adequate;
   b) None of the subjects involved had been exposed to program slicing or dicing before they made use of C-Sdicer; and
   c) Most of the subjects involved found slicing and dicing to be of use in debugging their programs.

Future Work

More extensive controlled experiments are needed to establish the usefulness of the C-Sdicer environment and its comparative merits and demerits with respect to other debugging approaches. The preliminary conclusions mentioned in the last section can serve as null hypotheses to be tested in the design of more comprehensive controlled experiments.

C-Sdicer could be enhanced to handle programs containing function calls and expressions involving pointers and bitwise operators so as to extend its range of popularity by including both novice and expert programmers. Slicing/dicing-based debugging tools could also be developed for other languages such as LISP and prolog. Slicing and dicing techniques could be incorporated into existing symbolic debuggers, which in turn would serve to enhance their debugging capabilities.

REFERENCES

[Barth 78] Barth, J. M., "A Practical Interprocedural Data Flow Analysis Algorithm", Communications of the ACM, vol. 21, 9, pp.724-736, September 1978.

[Conte 86] Conte, S. D., Dunsmore, H. E., and Shen, V. Y., Software Engineering Metrics and Models, The Benjamin/Cummings Publishing Company, Inc., CA, 1986.

[Hecht 77] Hecht, M. S., Flow Analysis of Computer Programs, North Holland, New York, 1977.

[Kernighan 78] Kernighan, B. W., and Ritchie, D. M., The C Programming Language, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Lyle 84] Lyle, J. R., Evaluating Variations on Program Slicing for Debugging, Ph.D. Dissertation, University of Maryland, College Park, MD, 1984.

[Weiser 81] Weiser, M., "Program Slicing", Proceedings of the Fifth International Conference on Software Engineering, pp. 439-449, March 1981.

[Weiser 82] Weiser, M., "Programmers Use Slices When Debugging", Communications of the ACM, vol. 25, 7, pp. 446-452, July 1982.

[Weiser 84] Weiser, M., "Program Slicing", IEEE Transactions on Software Engineering, vol. SE-10, 4, pp. 352-357, July 1984.

[Weiser 86] Weiser, M. and Lyle, J. R., "Experiments on Slicing-Based Debugging Aids", Empirical Studies of Programmers, Papers Presented at The First Workshop on Empirical Studies of Programmers, Soloway, E. and Iyengar, S., Editors, Ablex Publishing Corporation, Norwood, NJ, pp. 187-197, 1986.

FAULT DETECTION USING DATA FLOW TESTING
or
YOU SHOWED ME IT WAS CHEAP, BUT IS IT ANY GOOD?

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

(212) 998-3082
e-mail: weyuker@cs.nyu.edu

Abstract

We discuss a new empirical study aimed at evaluating both the cost and fault detection capabilities of various data flow testing strategies. Our findings corroborated earlier studies showing that the strategies were relatively cheap to use. We also present some surprising results about their fault detection capabilities.

Elaine Weyuker received a Ph.D. in Computer Science from Rutgers University, and an M.S.E. from the Moore School of Electrical Engineering, University of Pennsylvania. She is currently a Professor of Computer Science at New York University. Before coming to NYU, she was a programmer for Texaco, a system's engineer for IBM and was on the faculty of the City University of New York. Her research interests are in software engineering, particularly software testing and reliability, and software complexity measures. She is also interested in the theory of computation, and is the author of a book (with Martin Davis), "Computability, Complexity, and Languages", published by Academic Press.

She is currently the secretary/treasurer of ACM SIGSOFT, and is on the editorial board of ACM Transactions on Software Engineering and Methodology. She has been an ACM National Lecturer and was a member of the Executive Committee of the IEEE Computer Society Technical Committee on Software Engineering.

This research was supported in part by the National Science Foundation under grants CCR-85-01614 and CCR-89-20701, and by the Office of Naval Research under Contract N00014-85-K-0414.

A DEFINITION - CLEAR PATH WITH RESPECT TO VARIABLE x FROM NODE *i* TO NODE j IS A PATH (i, n_1, n_2, ... , n_m, j) SUCH THAT x HAS NO DEFINITIONS IN NODES $n_1, n_2, \ldots, n_m$

CLASSIFICATION OF VARIABLE OCCURENCES

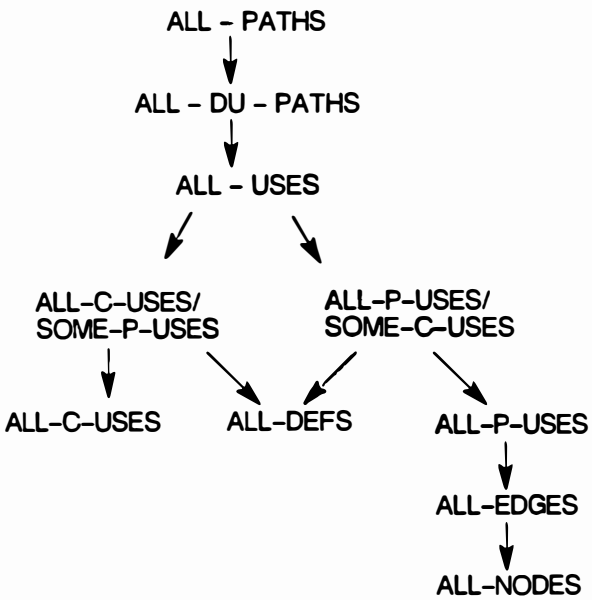
- DEFINITION - A VARIABLE IS GIVEN A VALUE
- PREDICATE USE (P-USE) - A VARIABLE IS REFERENCED IN A PREDICATE OF A DECISION STATEMENT
- COMPUTATION USE (C-USE) - A VARIABLE IS REFERENCED NOT IN A PREDICATE OF A DECISION STATEMENT

DATA FLOW TESTING CRITERIA

FOR EACH VARIABLE DEFINITION AT LEAST ONE DEFINITION-CLEAR PATH FROM THE DEF TO

{ALL / SOME} OF THE {P-USES / C-USES}

REACHABLE FROM THE DEF BY SOME DEFINITION-CLEAR PATH MUST BE EXECUTED.



GOALS OF THE STUDY

- HOW EXPENSIVE ARE THE DATA FLOW CRITERIA TO USE?

- MEANS OF PREDICTING THE NUMBER OF TEST CASES NEEDED TO SATISFY A GIVEN CRITERION FOR A GIVEN PROGRAM.

CRITERIA FOR SELECTING THE SOFTWARE

- REAL SOFTWARE

- WRITTEN BY PROFESSIONALS

- SUBSTANTIAL SIZE

METHODOLOGY

- USED 2 SUITES OF PROGRAMS

- TESTERS CHOSE TEST CASES TO DO A GOOD JOB OF TESTING SOFTWARE, NOT TO SATISFY CRITERIA.

- DATA FLOW CRITERIA USED TO ASSESS ADEQUACY.

- USED ATOMIC TEST CASES. NO ATTEMPT MADE TO MINIMIZE NUMBER OF TEST CASES.

- CONSIDERED PROGRAMS WITH 5 OR MORE DECISION STATEMENTS.

TABLE 2

average all-uses/all-defs	1.76
average all-du-paths/all-defs	1.96
average all-du-paths/all-uses	1.10

Least Squares

Number of Test Cases						
dec stmts	all-defs	all-edges	all-p-uses	all-c-uses	all-uses	all-du-paths
5	3	3	5	5	5	7
12	4	5	7	7	7	9
16	5	6	8	8	9	10

NOTATION

d - NUMBER OF (2-WAY) DECISION STATEMENTS IN SUBJECT PROGRAM

t - NUMBER OF TEST CASES SUFFICIENT TO SATISFY GIVEN CRITERION FOR SUBJECT PROGRAM

TABLE 1

	all-c-uses	all-p-uses	all-uses	all-du-paths
least squares	t = .36d + 2.82	t = .38d + 3.17	t= .39d + 3.76	t = .49d + 4.03
weighted average	t = .44d	t = .60d	t = .62d	t = .69d
maximum t/d	3.5	2.33	3.67	3.67

Effectiveness

all-defs	all-edges	all-p-uses	all-c-uses	all-uses	all-du-paths
31	32	32	33	33	34

Weighted Average

Number of Test Cases						
dec stmts	all-defs	all-edges	all-p-uses	all-c-uses	all-uses	all-du-paths
5	1	2	3	3	3	3
12	3	3	6	6	6	7
16	4	4	8	8	8	9

Unexecutable Associations

	no. associations	no. unexecutable	% unexecutable
all-defs	37	0	0
all-c-uses	71	8	11
all-p-uses	72	11	15
all-du-paths	182	102	56

Data-Flow Testing with Pointers and Function Calls

Thomas J. Ostrand
Siemens Corporate Research, Inc.
755 College Road East
Princeton, New Jersey 08540

e-mail: tjo@siemens.com

Abstract

Data flow analysis has previously been used to select and to evaluate test data for simple programs. We discuss some of the theoretical and implementation issues that arise in applying these methods to code with indirect memory references, and procedure calls.

Biography

Thomas J. Ostrand is a senior research scientist in the Intelligent Software Systems department of Siemens Corporate Research. He is currently the senior researcher for software testing work at SCR, and is involved in projects investigating both specification-based and program-based testing methods, as well as software maintenance methods for large, long-lived systems. He is currently directing a project to design and build a data-flow based testing system that handles aliasing and interprocedural data-flow in a realistic manner.

Dr. Ostrand was previously in the software research group of Sperry Univac, and a member of the computer science faculty at Rutgers University.

Data-Flow Testing with Pointers and Function Calls

Thomas J. Ostrand
Siemens Corporate Research
Princeton, New Jersey

Pacific Northwest Software Quality Conference
October 30-31, 1990
Portland, Oregon

Test Coverage Techniques

- **Specification-based**
- **Design-based**
- **Program-based**
  - **Control-flow**
    - **Statement testing**
    - **Branch testing**
    - **"Path testing"**
  - **Data-flow**
    - **All-defs**
    - **All-uses**
    - **All-DU-paths**

ASSET

A data-flow testing tool for Pascal programs.

ASSET handles almost all of Pascal, but certain features are treated in a simplistic manner:

- pointers -- treated as separate syntactic objects
- arrays -- entire array considered to be a single object
- interprocedural defs and uses:
  - all parameters considered uses and defs
  - interprocedural global variables ignored

A Data-Flow Tool for C

Goal: Build a data-flow testing tool that provides realistic treatment of

- pointers
- arrays
- function calls

Issues:

- Certain concepts must be modified to fit the C language
  - definition and use
  - def-clear path
  - the data-flow testing criteria
- Accurate static data-flow analysis is very difficult.
- Data-flow coverage monitor must check execution-time values of indirect references and function parameters.

Basic Data-Flow Concepts

Definition: An occurrence where a variable receives a value.

Use: An occurrence where a value is read from a variable.

Def-clear-path with respect to a variable V:

A sequence of nodes through the control flow graph, on which *V* is not assigned a value.

Def-use association (n,m,V):

- *V* is defined in *n*
- *V* is used in *m*
- At least one path from *n* to *m* is def-clear with respect to *V*

Applying the Concepts to C

Definite vs. Possible

Pointers with multiple aliases introduce static uncertainty about what is being referenced.

A def or use is *definite* if the program object being referenced is known at compile time.

Otherwise, the def or use is *possible.*

A path is *definite def-clear* with respect to a variable V if there are no definitions (definite or possible) of V on the path.

A path is *possible def-clear* if there are no definite definitions, and at least one possible definition of V on the path.

Def-Use Associations

Strong def-use association:

Every path is definite def-clear

Firm def-use association:

There is at least one definite def-clear path, but there is also at least one path that is either not def-clear or possible def-clear

Weak def-use association:

No paths are definite def-clear

Note that these classifications are based on static analysis of the code.

Exercising a Def-Use Association

Test case *t* exercises the def-use association (n,m,V) means that

When the program runs input *t* :

- **control reaches node n, and control subsequently reaches node m**
- **V is defined in node n**
- **V is used in node m**
- **V is not defined in any statement on the path executed from n to m**

The Data-Flow Criteria

All-defs: Exercise at least one def-use association for every def in the program

All-uses: Exercise every def-use association in the program

All-DU-paths: Exercise every def-use association (n,m,V), using every simple def-clear path from n to m

Problems for Data-Flow Analysis

- Determining which variable is being defined or used
- Determining if a given path is def-clear with respect to a given variable
- Avoiding spurious def-use associations
- Analyzing interprocedural data-flow

Problems in Analyzing C code

- Recognizing definitions and uses in "complex" C constructs
  - x++
  - x = (a > b) ? (x + y) : (y * z);
  - while (*s++ = *t++)

Problems for Interprocedural Analysis

- Capturing def-use associations for parameters
- Capturing def-use associations for global variables
- Carrying data-flow analysis across procedure boundaries

Solutions for Interprocedural Analysis

Parameters

- **Every actual parameter is considered *used* at call site.**
- **Pointer parameters**

  before call -- considered *use* of referenced variable

  after call -- considered *def* of referenced variable

Globals

- **used inside function -- considered *possible use* at call site**
- **defined inside function -- considered *possible def* at call site**

Monitoring Data-Flow Coverage

Pointer-free language:

Instrumenting each basic block is sufficient.

Language with pointers:

Reaching a basic block does not tell you whether a given definition or use is executed.

Solution:

Use a run-time tracing tool that monitors memory reads and writes, and reports actual memory addresses.

Consequence:

Run-time monitoring does not distinguish the three types of def-use associations.

Prototype Tool

Static Analysis

- **Transforms certain C constructs to analyzable form**
- **Builds program-point alias sets**
- **Produces static def-use associations**
- **Characterizes associations as strong, firm, or weak**

Prototype Tool

Dynamic Analysis

- **Monitors actual def-use associations**
- **Records test case execution paths**
- **Logs test cases and coverage produced by each case**
- **Evaluates program coverage in terms of all-defs, all-uses, (and all-DU-paths)**
- **Reports remaining unexercised def-use associations**
- **Runs interactively or in background**

Further Work

- **Refine the criterion hierarchy**
- **Sharpen static data-flow analysis algorithms**
- **Improve interprocedural flow analysis**
- **Construct visual, graphics-based user interface**

Verifying Software Quality Criteria Using an Interactive Graph Editor

Hausi A. Müller

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

Electronic mail: hausi@csr.uvic.ca
Phone: (604) 721-7630
Fax: (604) 721-7292

Keywords: Software quality, software metrics, encapsulation quality, partition quality, composition quality, subsystem compositions, graph editor, software maintenance, reverse engineering, design recovery.

Biographical sketch: Hausi A. Müller received a Diploma Degree in Electrical Engineering from the Swiss Federal Institute of Technology (ETH), Zürich in 1979 and the M.S. and Ph.D. degrees in computer science from Rice University, Houston, Texas in 1984 and 1986, respectively. From 1979 to 1982 he worked as a software engineer for Brown Boveri & Cie in Baden, Switzerland. He is currently an Assistant Professor of Computer Science at the University of Victoria, British Columbia. His research interests include software engineering, programming-in-the-large, programming environments, software metrics, graph algorithms, and computational geometry.

Verifying Software Quality Criteria Using an Interactive Graph Editor†

Hausi A. Müller

Department of Computer Science
University of Victoria
Victoria, B.C., Canada V8W 2Y2

Abstract

The software engineering principles *high strength within a subsystem, low coupling among subsystems,* and *small and few interfaces among subsystems* are often used as guidelines when constructing large software systems. In this paper, we define software quality criteria and measures to evaluate subsystem structures based on these principles. The quality measures quantify the *encapsulation effect* of individual modules or subsystems as well as the effectiveness of module or subsystem compositions with respect to *separation of concerns.* Algorithms which compose subsystem structures according to these measures are an integral part of our Rigi system — an interactive graph editor for the presentation and representation of the structure of large software systems. We show how the graph editor can be used to analyze and verify the quality of subsystem structures with respect to the defined module and subsystem interconnection criteria.

Keywords: Software quality, software metrics, encapsulation quality, partition quality, composition quality, subsystem compositions, graph editor, software maintenance, reverse engineering, design recovery.

1. Introduction

A primary aim of software engineering research is to devise and perfect methodologies and tools for producing quality software. While there are many facets that make up the quality of a software system, this paper focuses on programming-in-the-large software quality criteria in general and subsystem (de)composition quality criteria in particular.

To manage and master their complexity, large software systems are usually decomposed into a hierarchy of modules and subsystems. Software engineers use software engineering principles such as *high strength within a subsystem* and *small and few interfaces among subsystems* to guide the decomposition process. However, the clustering or partitioning of components into subsystem structures is typically not supported by an automatic procedure and therefore performed manually. Moreover, the quality of the resulting structures is rarely measured and evaluated.

This paper offers a metric for automatically quantifying the quality of subsystem structures. The first part of the paper presents a rigorous definition of a composition quality measure based on the software engineering principles *information hiding* and *separation of concerns.* The quality of a composition is defined as a function of the amount of information encapsulated or abstracted by its individual com-

† This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, the British Columbia Advanced Systems Institute, and the IRIS Federal Centre of Excellence.

ponents and the strength of the coupling among its components. The second part of the paper outlines a methodology for constructing and evaluating subsystem structures using an interactive graph editor.

2. Related work

Several metrics for assessing the quality or complexity of software structures have previously been proposed by McCabe [McCa 76], Halstead [Hals 77], DeMarco [DeMa 82], as well as Howatt and Baker [HoBa 89]. While all these metrics deal with software structure, they measure programming-in-the-small features such as the number of paths through a program [McCa 76] or nesting levels [HoBa 89]. The composition metric presented here is specifically designed to measure the "programming-in-the-large quality" of software structure.

In a recent paper, McCabe and Butler defined three software design metrics [McBu 89] — *module design complexity*, *design complexity*, and *integration complexity*. They define the design complexity of a module as the *cyclomatic complexity* [McCa 76] of its *reduced flow graph* — the nodes and edges that possibly affect module interrelationships. The design complexity of a module composition is then simply the sum of its module complexities. This metric is definitely a step toward measuring the "programming-in-the-large quality" of a software system.

Our *encapsulation* and *composition* quality measures, presented in Section 5, roughly correspond to their module design and design complexity measures. However, our measures not only apply to module graphs, but also to subsystem compositions. By computing the cyclomatic complexity for reduced flow graphs instead of complete flow graphs, McCabe and Butler eliminate some of the immaterial programming-in-the-small complexity. However, our software quality measures further emphasize the programming-in-the-large aspects: our composition measure not only takes the qualities of the individual components into account as in the McCabe and Butler metric, but also measures the separation or firewall effect of the components involved.

3. Software structure models

Software structures such as control flow, data flow, and resource flow are often modeled by directed weighted graphs. We use a special class of layered graphs for presenting and representing software systems. The layers represent resource relations and the edges between adjacent layers represent composition relations.

3.1. Resource relations

The primary models used to describe, represent, and manage software structure are the *unit interconnection model* and the *syntactic interconnection model* [Perr 87].

It is convenient for us to think of such an interconnection model as a directed weighted *resource-flow graph* (RFG). The vertices and edges of an RFG represent system components and the dependencies induced by their resource supplier-client relation. A directed edge from node a to b indicates that module a provides a set of syntactic objects to module b. Depending on the application, the edge weights are a list of resource names (e.g., $S = \{\alpha, \beta, \gamma, \delta\}$), the cardinality of the resource set (e.g., $|S| = 4$), or even absent. The main distinction between the unit and the syntactic models is the granularity of interconnection ranging from *files*, *subsystems*, and *modules* in the unit model to nameable entities defined in a programming language — *procedure*, *constant*, *type*, and *variable* — in the syntactic model.

3.2. Composition relations

The graphs induced by the resource supplier-client relation are "flat" and typically unwieldy. To add organizational axes, composition relations are imposed on these graphs. A composition relation collapses resource-dependency subgraphs to form a sequence of RFG layers $G_1, ..., G_n$. The edges between two adjacent layers in such a graph represent composition dependencies whereas the edges within a layer represent resource dependencies. The grain size of the nodes generally increases with the layer number while the size of the interfaces among the nodes decreases.

3.3. Exact interfaces

To compose, analyze, and evaluate subsystem structures according to software engineering principles such as small interfaces, we not only need component dependency information but also the exact number and kinds of resources exchanged among the components of a software system.

At the unit level, most programming languages are imprecise with respect to requisition and provision of resources [WoCW 88]. Modules import and export entire interfaces rather than specific objects. However, by parsing the source text or by inspecting the cross-reference listings produced by the compiler, precise import and export lists or *exact syntactic interfaces* can be computed.

It is useful to investigate how a specific subsystem relates to the remainder of the entire system and how it relates to other specific subsystems. Thus, we distinguish between an *exact component interface* and an *exact dependency interface*. A component interface specifies how a subgraph of an RFG relates to the remainder of the graph whereas a dependency interface specifies how two RFG subgraphs depend on each other.

Definition. Let V_s and V_t be two node subsets of an RFG. Then the following equations define the sets of *exact provisions* P and *exact requisitions* R of V_s with respect to V_t in terms of component provisions and requisitions.

$$P_d(V_s \mid V_t) = \bigcup_{\substack{v \in V_s \\ w \in V_t}} \Big(P_c(v) \cap R_c(w) \Big)$$

$$R_d(V_s \mid V_t) = \bigcup_{\substack{v \in V_s \\ w \in V_t}} \Big(R_c(v) \cap P_c(w) \Big)$$

The *exact dependency interface* between V_s and V_t consists of exactly those resources that are exchanged between V_s and V_t.

$$E_d(V_s, V_t) = P_c(V_s \mid V_t) \cup R_d(V_s \mid V_t)$$

The *exact component interface* of a node subset V_s consists of those resources that are exchanged between V_s and all other nodes in the system $V - V_s$.

$$P_c(V_s) = P_d(V_s \mid V - V_s)$$

$$R_c(V_s) = R_d(V_s \mid V - V_s)$$

$$E_c(V_s) = P_c(V_s) \cup R_c(V_s)$$

□

Note that the following identities hold.

$$P_d(V_s \mid V_t) = R_d(V_t \mid V_s)$$

$$R_d(V_s \mid V_t) = P_d(V_t \mid V_s)$$

$$E_d(V_s, V_t) = E_d(V_t, V_s)$$

$P_d(V_s \mid V_t)$ is the intersection of the set of objects defined in V_s and the set of object referenced in V_t; $R_d(V_s \mid V_t)$ is the intersection of the set of objects referenced in V_s and the set of object defined in V_t.

The exact interfaces of a particular level in a layered subsystem composition can be computed by inspecting the dependencies of the next lower level or by parsing the source text.

Note that the exact provisions of a component are often a subset of the objects provided by the component. Consequently, subsystems can encapsulate large interfaces, providing a considerably smaller set

of objects to the remainder of the system (i.e., a *small interface*).

4. Composition measures

This section introduces two pairs of software composition measures for RFGs. The purpose of the first pair is to capture the two software engineering principles *high strength within a component* and *low coupling among components*. The intention of the second pair is to identify loosely coupled components having *common clients* or *common suppliers*. The latter measure satisfies the software engineering principle *few interfaces*, because merging neighbors with common client/supplier reduces the number of interfaces among the components involved.

The measures presented in this section constitute the foundation of both our composition methodology [MüUh 90] and our composition metric.

4.1. Interconnection strength measure

Definition. The *interconnection strength* $IS(v, w)$ of two nodes, v and w, in an RFG is equal to $|E_d(v, w)|$, the cardinality of the exact dependency interface between v and w (i.e., the exact number of syntactic objects exchanged between the two nodes). Two components are said to be *strongly coupled* iff their interconnection strength is greater than a certain threshold T_h and *loosely coupled* iff their interconnection strength is less than a certain threshold T_l. □

Definition. The *centricity* $CT(v)$ of a node v in an RFG is equal to $|E_c(v)|$, the cardinality of the exact component interface of v (i.e., the exact number of syntactic objects exchanged between v and all other nodes in the RFG). A node v is said to be a *central* or *key component* iff its centricity is greater than a certain threshold T_k and a *fringe component* iff its centricity is less than a certain threshold T_f. □

Definition. Let $G = (V, E)$ be a weighted RFG where the edge weights, denoted by w, represent the exact number of syntactic objects exchanged among the components. Then the following equations summarize the *high-* and *low-strength interfaces* of G.

$$I_h(G) = \{\, e \in E \mid w(e) \geq T_h \,\}$$

$$I_l(G) = \{\, e \in E \mid w(e) \leq T_l \,\}$$

$$N_h(G) = |I_h(G)|$$

$$N_l(G) = |I_l(G)|$$

The sets $I_h(G)$ and $I_l(G)$ represent the high- and low-strength edges of G; $N_h(G)$ and $N_l(G)$ denote the number of edges in $I_h(G)$ and $I_l(G)$, respectively. □

4.2. Common client/supplier measure

Definition. Let $SUCC(v)$ denote the direct clients (immediate successors) and $PRED(v)$ the direct suppliers (immediate predecessors) of a node v in an RFG $G = (V, E)$. Then two components are similar with respect to their clients iff they provide objects to a common subset of clients. Analogously, two nodes are similar with respect to their suppliers iff they require objects from a common subset of suppliers. The common client and supplier subsets of a subset $M \subseteq V$ of components, $C(M)$ and $S(M)$, respectively, are defined by the following set equations.

$$C(M) = \bigcap_{v \in M} SUCC(v)$$

$$S(M) = \bigcap_{v \in M} PRED(v)$$

The elements of M are said to be *neighbors* or *siblings* with respect to their clients (suppliers) iff the cardinality of their client (supplier) subset $|C(M)|$ ($|S(M)|$) is greater than a certain threshold T_c (T_s).

Let I_c (I_s) denote all those sets of components that have at least T_c (T_s) common clients (suppliers).

$$I_c(T_c) = \{ M \subseteq V \mid |C(M)| \geq T_c \}$$

$$I_s(T_s) = \{ M \subseteq V \mid |S(M)| \geq T_s \}$$

By collapsing each element of I_c (I_s) into a subsystem, the number of edges or interfaces among nodes is reduced by N_c (N_s).

$$N_c(T_c) = \sum_{x \in N_c} \left((|x| - 1) \cdot |C(x)| \right)$$

$$N_s(T_s) = \sum_{x \in N_s} \left((|x| - 1) \cdot |S(x)| \right)$$

□

5. Assessing the quality of compositions

When composing subsystem structures manually, software engineers make intuitive or subjective decisions based on experience, skill, and insight. Building subsystems structures automatically by interconnection strength or common clients/suppliers makes the process more objective with respect to a given similarity measure.

We propose the following three *software quality measures* to compare the quality of individual subsystems and subsystem compositions with respect to certain software engineering principles. The first quality measure quantifies the *encapsulation* effect of a particular module or subsystem. The second one measures the *partition* effect of a subsystem composition.* Finally, these two measures are combined yielding a composition quality measure. Component coupling — the number and the size of the interfaces among the components — is the basis for all three quality measures. The measures are normalized so that their values are in the range zero (lowest quality) through one (highest quality).

5.1. Encapsulation quality

Definition. Let $G = (V, E)$ be a subgraph of an RFG. Then the *normalized encapsulation quality* EQ of G is defined as

$$EQ(G) = \begin{cases} 1 & \text{if } G \text{ is a singleton node} \\ \dfrac{N_h(G) + N_c(G) + N_s(G)}{e + |E_c(G)| - 1} & \text{otherwise} \end{cases}$$

where $N_h(G)$ is the number high-strength interfaces in G; $N_c(G)$ and $N_s(G)$ are the number of interfaces absorbed due to the common clients and suppliers of G; e is the number of edges in G; $|E_c(G)|$ is the number of interfaces between G and the remaining graph; and the purpose of the denominator is to normalize the encapsulation quality measure; $e + |E_c(G)| - 1$ is an upper bound for the number of high-strength and client/supplier interfaces that can possibly be internalized and absorbed by collapsing G. □

Thus, the information hiding or encapsulation quality of a subsystem is defined so as to increase with the number of its high-strength interfaces and the number of its absorbed client/supplier interfaces. As a result, if the components of a subsystem are cohesive or tightly coupled, its encapsulation quality is *high*; analogously, if the components are largely independent or loosely coupled, the encapsulation quality is *low*.

* In the software engineering literature, the terms *cohesion* and *coupling* are traditionally used to describe the encapsulation and partition effect, respectively. We introduced our own terms, because our metrics only address a small subset of the meanings and interpretations that are associated with the terms cohesion and coupling.

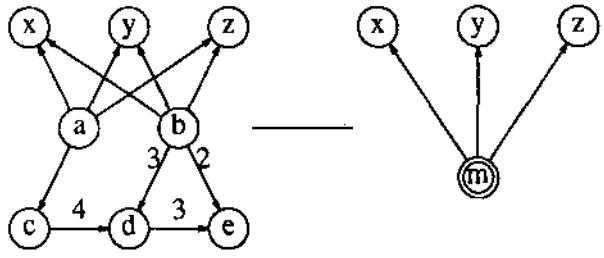


Figure 1. Encapsulation quality

As an example, consider the aggregate subsystem m in Figure 1 consisting of components a, b, c, d, and e and assume the thresholds $T_h = 2$ and $T_c = 3$. Since components a and b have three clients in common (i.e., x, y, and z), aggregating a, b into m reduces the number of interfaces between the node subsets $\{ a, b \}$ and $\{ x, y, z \}$ by three. Moreover, the number of high-strength interfaces in m is 4 (unlabeled edges have a weight of one). Subsystem m internalizes five edges and there are six edges between m and the remaining graph. Thus, $e = 5$ and $|E_c(m)| = 6$ and, hence, the normalized encapsulation quality of subsystem m is

$$EQ(m) = (4 + 3) / (5 + 6 - 1) = 0.7$$

5.2. Partition quality

Definition. Let $G = (V, E)$ be a weighted RFG. Then the *normalized partition quality* PQ of G is defined as

$$PQ(G) = \frac{1}{1 + \dfrac{N_l(G) + n \cdot N_h(G)}{n(n-1)}}$$

where n is the number of nodes in G; $N_l(G)$ and $N_h(G)$ are the numbers of low- and high-strength interfaces in G, respectively. □

Thus, the partition quality of a modularization or subsystem (de)composition increases with the number of its small interfaces and the sparsity of its graph; PQ decreases with the number of large interfaces and the density of G. Note that the defined partition quality is bounded below by zero and above by one.

The above equation is finely tuned so that the partition quality measure produces meaningful results even for extreme composition structures. The following examples illustrate the behavior of the partition metric for some extreme cases. The number of components or nodes, n, is assumed to be positive.

(1) A tree G_t with $n = 10$ nodes, $n - 1$ loosely coupled interfaces, and zero strongly coupled interfaces is considered an ideal composition structure.

$$PQ(G_t) = \frac{1}{1 + \dfrac{1}{n}} = \frac{n}{n+1} = 0.91$$

(2) In contrast, a complete graph G_k with $n(n-1)$ loosely coupled directed interfaces and zero strongly coupled interfaces is considered a mediocre partition despite its many small interfaces.

$$PQ(G_k) = \frac{1}{1 + \frac{n(n-1)}{n(n-1)}} = 0.5$$

(3) Another extreme, but ideal, composition, is a graph G_d of n disconnected components (i.e., G_d has no interfaces and, hence, $N_l(G) = N_h(G) = 0$).

$$PQ(G_d) = \frac{1}{1 + \frac{0}{n(n-1)}} = 1$$

(4) As a final example, consider a typical high-quality subsystem resource-flow graph G_r consisting of nine subsystems, eight loosely coupled interfaces, and two strongly coupled interfaces.

$$PQ(G_r) = \frac{1}{1 + \frac{8 + 9 \cdot 2}{9 \cdot 8}} = 0.73$$

5.3. Composition quality

Definition. Let $G = (V, E)$ be a weighted RFG. Then the *normalized composition quality* CQ of G is defined by the following equation

$$CQ(G) = \frac{n \cdot PQ(G) + \sum_{m \in V} EQ(m)}{2n}$$

where n is the number of nodes in G; $PQ(G)$ is the normalized partition quality of G; and $EQ(m)$ is the normalized encapsulation quality of a node m in G. □

Thus, the composition quality of a module or subsystem (de)composition increases with the cumulative encapsulation quality of its components and with its partition quality. We assume that information hiding and separation of concerns are equally important objectives when designing module or subsystem compositions.

As an example, consider the graph $G = (V, E)$ depicted in Figure 2. The node labels represent encapsulation qualities and the edge labels denote interface sizes. Assuming $T_l = 1$ and $T_h = 3$, G exhibits three small interfaces (i.e., $w(e) \leq T_l$), one large interface (i.e., $w(e) \geq T_h$), and one medium size interface (i.e., $T_l < w(e) < T_h$) among its components. Thus, the encapsulation, partition, and composition qualities of G are computed as follows:

$$\sum_{m \in V} EQ(m) = 0.3 + 0.3 + 0.7 + 0.2 + 0.5 = 2.0$$

$$PQ(G) = \frac{1}{1 + \frac{3 + 5}{20}} = 0.71$$

$$CQ(G) = \frac{5 \cdot 0.71 + 2.0}{2 \cdot 5} = 0.55$$

6. Rigi editor

The Rigi editor is a distributed graph and hierarchy editor which allows the users to edit, maintain, and explore the objects stored in an underlying graph data base [MüKl 88]. This section outlines the salient operations of the graph editor for composing, manipulating, and evaluating subsystem structures.

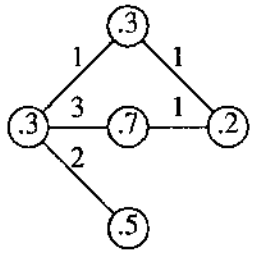


Figure 2. Partition and composition quality

6.1. Basic graph editing

Entire subgraphs and hierarchies of subgraphs may be duplicated and deleted using the operations *Cut*, *Copy*, *Paste*, and *Clear*. For example, applying the copy operation to a subsystem node involves the copying of all objects and dependencies associated with that subsystem including modules, interfaces, versions, and source documents. Thus, the description of an entire software system, or a subset thereof, can be duplicated with the touch of a mouse button.

6.2. Iterators

The Rigi system provides operational abstractions called *iterators* to traverse the layered graphs of the underlying data base. An iterator performs the same operation on every object of a collection of elements such as a list of nodes or a graph of nodes and edges.

There are list iterators to traverse the elements in a given set (*e.g.*, the nodes or the edges of a layer). Tree and graph traversals are implemented using *depth first search* and *breadth first search iterators*. Thus, to perform an operation on the nodes and edges that are reachable from a given node, one simply applies a tree iterator to that node.

6.3. Searching

The Rigi editor includes *zooming*, *searching*, *clipping*, and *filtering* features which allow the users to swiftly navigate through the myriad objects and dependencies and to quickly identify pertinent information.

The search commands include "grepping" (i.e., performing a string search using regular expressions as supported by the Unix operating system) for string patterns in node and edge labels. Filters and iterators are used to guide the searching. For example, the search space can be restricted to a fixed set of node classes (*e.g.*, modules and subsystems).

The iterators can be programmed to search a variety of subgraphs. When composing subsystem structures, nodes are often aggregated by name. Nodes with similar names can be identified by searching the labels of all nodes in a graph for a given regular expression.

6.4. Filtering and Clipping

Rigi offers *node* and *edge class filters* for focusing on individual semantic networks or contexts (i.e., a graph whose nodes and edges are members of a single node and/or edge class, respectively) or combinations of individual contexts. Thus, one can easily focus on the dependencies of a certain class of nodes (e.g., *include files*) by invoking a node class filter.

Another powerful way of constraining search spaces is filtering through *clipping*. Nodes that are immaterial for system comprehension or irrelevant for the current investigation can be clipped (i.e., removed from the current graph).

The clipping and filtering operations do not affect the underlying object base. They simply allow the user to impose different views on the entities of the repository.

6.5. Collapsing

Collapsing is a particularly useful graph transformation for defining and composing subsystem structures. The operation *collapse* essentially replaces a subgraph — a set of functions, modules, or subsystems — by a single node — a subsystem. Its inverse restores the original graph. To make this operation completely reversible, not only the subgraph, but also the edges between the subgraph and the remaining graph, are restored.

6.6. Composition Operations

The Rigi editor provides a set of composition operations based on the measures presented in Section 4. Nodes are identified interactively using searching mechanisms and by adjusting thresholds. If the nodes are sufficiently related, they are selected and can then be collapsed to form a subsystem. The most important compositions are by name, by sibling, by interconnection strength, and by centricity.

Composition by name

Nodes with similar name patterns can be identified using the search facilities of the editor.

Composition by sibling

Nodes such as library routines are often related because they provide objects to similar sets of clients or require objects from similar sets of suppliers. Thus, the editor provides operations to identify nodes that have *common clients* or *suppliers*. The size of the sets is determined by the thresholds T_c and T_s.

Composition by interconnection strength

Strongly coupled and loosely coupled components are identified by varying the thresholds T_h and T_l. Strongly coupled components are assigned to the same subsystem and loosely coupled ones are assigned to different subsystems.

Centricity

By adjusting the thresholds T_k and T_f, central and fringe component nodes can be identified.

These steps constitute the core of our subsystem composition methodology [MüUh 90]. They can be applied in any order and to any level in a subsystem hierarchy. However, composition by sibling is typically applied to construct low-level subsystems, followed by composition by name to form the middle part of the hierarchy, and finally composition by interconnection strength and centricity to build the most abstract subsystems.

6.7. Quality measures

The software quality measures — encapsulation, partition, and composition quality — can be computed for any selection of nodes as described in Section 5 and are currently output to a text window. Since the measures are normalized, the values computed for different subgraphs can be readily compared. In particular, the values can be used to guide the subsystem identification process, to compare existing and newly constructed modularizations, or to assess the quality of entire systems. The absolute values of the measures should always be interpreted with caution. Nevertheless, if the composition quality of an RFG is close to zero, then the RFG is probably not designed by adhering to the software engineering principles information hiding and/or separation of concerns. Moreover, a low partition quality value of an RFG indicates a lack of firewalls.

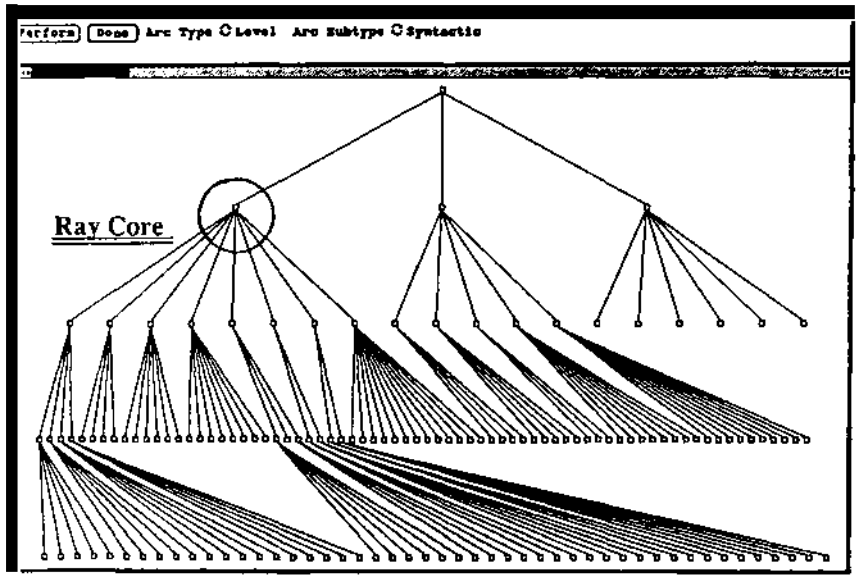


Figure 3. Subsystem hierarchy of ray tracing rendering system

7. Analyzing and verifying the quality of software structure

Quality software should have the property of being understandable and measurable [Adri 82]. Using the composition facilities provided by the Rigi editor, software structures such as call graphs, module graphs, include file dependency graphs, and directory hierarchies can be summarized, analyzed, evaluated, and optimized subject to software engineering principles. Being able to retrieve, browse, and trace these structures effectively is a key to software comprehension particularly during the maintenance phase. The programming-in-the-large quality measures can be used to evaluate and compare alternatives of these structures. The constructed composition structures as well as their associated quality measures can also serve as management tools when estimating the complexity of development, integration, testing, and maintenance tasks.

We now briefly outline what steps a software engineer might perform when analyzing the structure of a software system using Rigi. One of the systems we analyzed over and over again, while developing and perfecting the subsystem composition methodology and the programming-in-the-large metrics, was a ray tracing rendering system [Corr 90].

To generate an initial RFG, we first parse the source code of the ray tracer to extract the function dependencies automatically.* The renderer consists of 109 C functions and 172 call dependencies among these functions.** Using the composition operations provided by the Rigi editor, subsystem hierarchies are then built interactively on top of the flat function graph. The thresholds values can be combined in numerous ways to obtain a variety of graph structures. As a result, the user has to choose among a huge collection of subsystem structure alternatives.

* The Rigi system can also be configured so that it not only extracts the function dependencies, but also the module graph, and the directory structure. In this case, the user has a hierarchy to work with rather than a flat graph.

** The number of edges is low compared to the number of nodes, because of two reasons: (1) some debugging routines are included in the system, but they are currently not referenced by any of the routines in the system; and (2) some of the functions are invoked through function pointers; function pointer dependencies are currently not recognized by our parser.

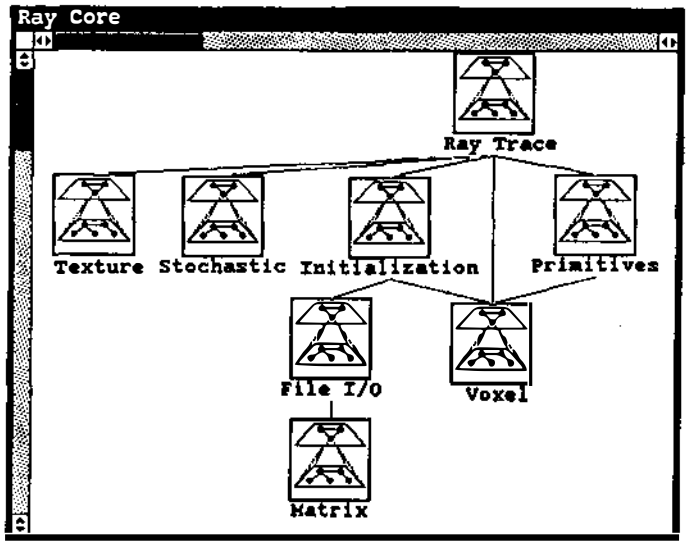


Figure 4. The heart of the ray tracer

When discriminating among alternatives, one can evaluate them manually by inspecting their visual representations or automatically by computing their normalized composition qualities. When the screen is covered with hundreds of nodes and dependencies, it is cumbersome to compare alternatives visually. However, it is easy to identify a set of nodes using the composition operations and to compute its normalized encapsulation, partition, and composition qualities. The rough compositions can then be finely tuned by adjusting individual nodes and edges while observing the effect on the quality measures.

A composed subsystem hierarchy of the ray tracer is depicted in Figure 3. Using the Rigi editor, it takes the designer of the ray tracer approximately 15 minutes to reconstruct this hierarchy from the flat function graph. The lower levels are built using composition by name and the higher-order subsystems are aggregated according to their interconnection strength. A software engineer familiar with the Rigi editor, but unfamiliar with the ray tracer, can build a comparable structure in about an hour. The graph depicted in Figure 4 represents the structure of the core subsystem of the ray tracer. It corresponds to the circled node in Figure 3 and is thus a high-level subsystem. The composition qualities associated with this subsystem are shown in Figure 5. The encapsulation qualities of most subsystems are low, because their components were grouped by name rather than interconnection strength. Although there are more high-strength interfaces than low-strength interfaces, the partition quality is fairly high because the system is almost tree structured.

Composition Qualities of Subsystem Ray Core					
Thresholds		**Encapsulations**		**Qualities**	
		EQ(Ray Trace)	0		
n	8	EQ(Texture)	0.5		
e	9	EQ(Voxel)	0	EQ	0.1
T_h	3	EQ(File I/O)	0.3	PQ	0.6
T_l	1	EQ(Matrix)	0	CQ	0.3
N_h	4	EQ(Stochastic)	0		
N_l	3	EQ(Init)	0		
		EQ(Primitives)	0		

Figure 5. Composition quality measures

8. Conclusions

Using the composition methodology and the procedures for evaluating subsystem, partition, and composition structures, software engineers can devise cost-effective testing, integration, and maintenance strategies. It is too early for us to realize the full potential of our composition metrics, but we are confident that the measures can be tailored to produce viable time and cost estimates for maintenance tasks.

Acknowledgements

The author would like to thank Brian Corrie, Nazim Madhavji, Paul Sihota, Craig Sinclair, Lee Thomas, Jim Uhl, and an anonymous referee for numerous suggestions that improved both the metrics and this presentation. Brian integrated the software quality metrics described in this paper into our Rigi system.

References

[Adri 82] Adrion, W.R., *et al.* "Validation, verification, and testing computer software," *ACM Computer Surveys*, 14(2), June 1982.

[Corr 90] Corrie, B. "A Workbench for Realistic Image Synthesis," M.Sc. Thesis, University of Victoria, April 1990.

[DeMa 82] DeMarco, T. *Controlling Software Projects: Management, Measurement & Estimation*, Yourdon Press, 1982.

[Hals 77] Halstead, M.H. *Elements of Software Science*, Elsevier, 1977.

[HoBa 89] Howatt, J.W. and A.L. Baker. "Rigorous Definition and Analysis of Program Complexity Measures: An Example Using Nesting," *The Journal of Systems and Software*, Elsevier, 10, pp. 139-150, 1989.

[McBu 89] McCabe, T.J. and C.W. Butler. "Design Complexity Measurement and Testing," *Communications of the ACM*, 32(12), pp. 1415-1425, December 1989.

[McCa 76] McCabe, T.J. "A Complexity Measure," *IEEE Transaction on Software Engineering*, SE-2(4), pp. 308-320, December 1976.

[MüKl 88] Müller, H.A; and K. Klashinsky. "Rigi — A System for Programming-in-the-large," In *Proceedings of the 10th International Conference on Software Engineering (ICSE)*, (Raffles City, Singapore, IEEE Computer Society Press (Order Number 849), April 11-15), pp. 80-86, April 1988.

[MüUh 90] Müller, H.A; and J.S. Uhl. "Composing Subsystem Structures Using (K,2)-Partite Graphs," To appear in proceedings of *Conference on Software Maintenance — 1990.* (San Diego, CA, November 26-29), IEEE Computer Society Press, 1990.

[Perr 87] Perry, D.E. "Software Interconnection Models," In *Proceedings of the 9th International Conference on Software Engineering*, (Monterey, CA, March 30 - April 2), pp. 61-69, April 1987.

[WoCW 88] Wolf, A.L.; L.A. Clarke; and J.C. Wileden. "A Model of Visibility Control," *IEEE Transactions on Software Engineering*, SE-14(4), pp. 512-520, April 1988.

Appendix — Glossary of symbols

RFG	resource-flow graph
V	set of $n = \|V\|$ nodes
E	set of $e = \|E\|$ edges
$w(x)$	weight of edge x

$SUCC(x)$ set of successor nodes of node x

$PRED(x)$ set of predecessor nodes of node x

G resource-flow graph $G = (V, E)$

V_s subset of the nodes of an RFG

$P_c(V_s)$ exact provisions of subsystem V_s

$R_c(V_s)$ exact requisitions of subsystem V_s

$E_c(V_s)$ exact component interface of V_s

$P_d(V_s|V_t)$ exact provisions of subsystem V_s with respect to subsystem V_t

$R_d(V_s|V_t)$ exact requisitions of subsystem V_s with respect to subsystem V_t

$E_d(V_s|V_t)$ exact dependency interface of V_s with respect to subsystem V_t

$IS(v, w)$ interconnection strength, the cardinality of the exact dependency interface between v and w

$CT(v)$ centricity, the cardinality of the exact component interface of v

T_l threshold for identifying loosely coupled components

T_h threshold for identifying strongly coupled components

T_k threshold for identifying central or key components

T_f threshold for identifying fringe components

T_c threshold for identifying common clients

T_s threshold for identifying common suppliers

$I_l(G)$ set of low-strength interfaces in G

$I_h(G)$ set of high-strength interfaces in G

$N_l(G)$ number of low-strength interfaces in G

$N_h(G)$ number of high-strength interfaces in G

M subset of the nodes of an RFG which are neighbors or siblings (i.e., they common clients and/or suppliers)

$C(M)$ all nodes in M have $C(M)$ clients in common

$S(M)$ all nodes in M have $S(M)$ suppliers in common

$I_c(T_c)$ the family of common client subsets of subgraph M; each family member has at least T_c clients

$I_s(T_s)$ the family of common supplier subsets of subgraph M; each family member has at least T_s clients

$N_c(T_c)$ the number of edges absorbed due to at least T_c common clients when collapsing the elements of N_c

$N_s(T_s)$ the number of edges absorbed due to at least T_s common suppliers when collapsing the elements of N_c

$EQ(G)$ normalized encapsulation quality of G

$PQ(G)$ normalized partition quality of G

$CQ(G)$ normalized composition quality of G

Visualization of Quality Factors in Software Construction

Ilkka Tervonen
Department of Information Processing Science
University of Oulu
SF-90570, Oulu
Finland

Abstract

Four aspects of the visualization of quality factors in software construction are considered. Visualization is illustrated with a prototype of the Quality Game, the purpose of which is to introduce novice designers to the world of quality assurance. The Quality Game visualizes the use of quality factors in software development, the correlation between quality and resources, and the usability of quality factors, criteria and checklists for software validation.

Biographical sketch

Ilkka Tervonen received a Lic. Phil. degree from the University of Oulu, and is currently a Ph.D. candidate of the same University. His research interests include conceptual modelling of software engineering, software quality assurance and Smalltalk programming. He is currently an acting Associate Professor of Information Processing Science at the University of Oulu.

Visualization of Quality Factors in Software Construction

Ilkka Tervonen
Department of Information Processing Science
University of Oulu
SF-90570, Oulu
Finland

1. INTRODUCTION

Software quality, like quality generally, is very difficult to capture and define, and consequently also very difficult to teach, and the situation is very similar as regards learning programming. In this context, the Quality Game has connections with two branches of research: tutoring systems and program visualization. Our first prototype does not have characteristics of an intelligent tutoring system (ITS) such as student modelling or explanation (Clancey 1987), but it is more akin to the idea of a "microworld", presented by Papert (1980), in which a child or novice student can proceed safely by incremental steps and learn new concepts of the subject. This also provided one stimulus for the present experiment of building an environment, in which novice designers can train in perceiving the impacts of quality factors on software construction.

Program visualization and visual programming (Shu 1988) have been the most popular visualization topics in software engineering, as is natural, because code is in any case the most concrete object in software construction. This also serves to warn us about the difficulties of visualizing such an abstract issue as quality factors in software construction. We realize that there are some very strong limitations to the teaching of software quality and quality factors. We can teach only main structures and principles, because we must redefine the relevant factors, criteria and checklists separately for each prepared example.

The purpose has been to create a training environment which simulates the operations that take place in a normal software project, i.e. we will consider here the use of quality factors, criteria and checklists in software development, the correlation between quality and resources and the validation of software using quality factors, criteria and checklists. The main objective of the Quality Game is to form an idea of these quality issues, so that when the designer meets up with them in a real project he/she will not be totally confused.

2. ASPECTS OF VISUALIZATION

The Quality Game ties in three research branches in Software Engineering (SE), namely Conceptual modelling, Object-oriented software construction and Quality assurance. Conceptual modelling is based on the OCT (O = Organization, C = Conceptual, T = Technical) framework (Iivari 1989), which defines three levels of abstraction with some sub-levels and three perspectives (structure, function and behaviour). The perspectives are commonly accepted in the SE area (c.f. RT-SA/SD method, Ward and Mellor 1985), but the abstraction levels are not as familiar to ordinary designers. Each level of abstraction focuses on a specific viewpoint, i.e. decisions are based on organizational impact at the O level, user satisfaction at the C level and reusability and efficiency at the T level. Due to the important role of the framework, its levels of abstraction and their connection with quality

assurance also form an aspect to be visualized by the Quality Game (Aspect 1 in Figure 2). The other aspects are justified by the Universe of Discourse description (also a sub-level in the OCT framework), which defines the world behind the game, i.e. Quality Assurance in a specific company (Figure 1). Because levels of abstraction are in themselves too general, we must apply them to some development methodology (e.g. object-oriented development methodology).

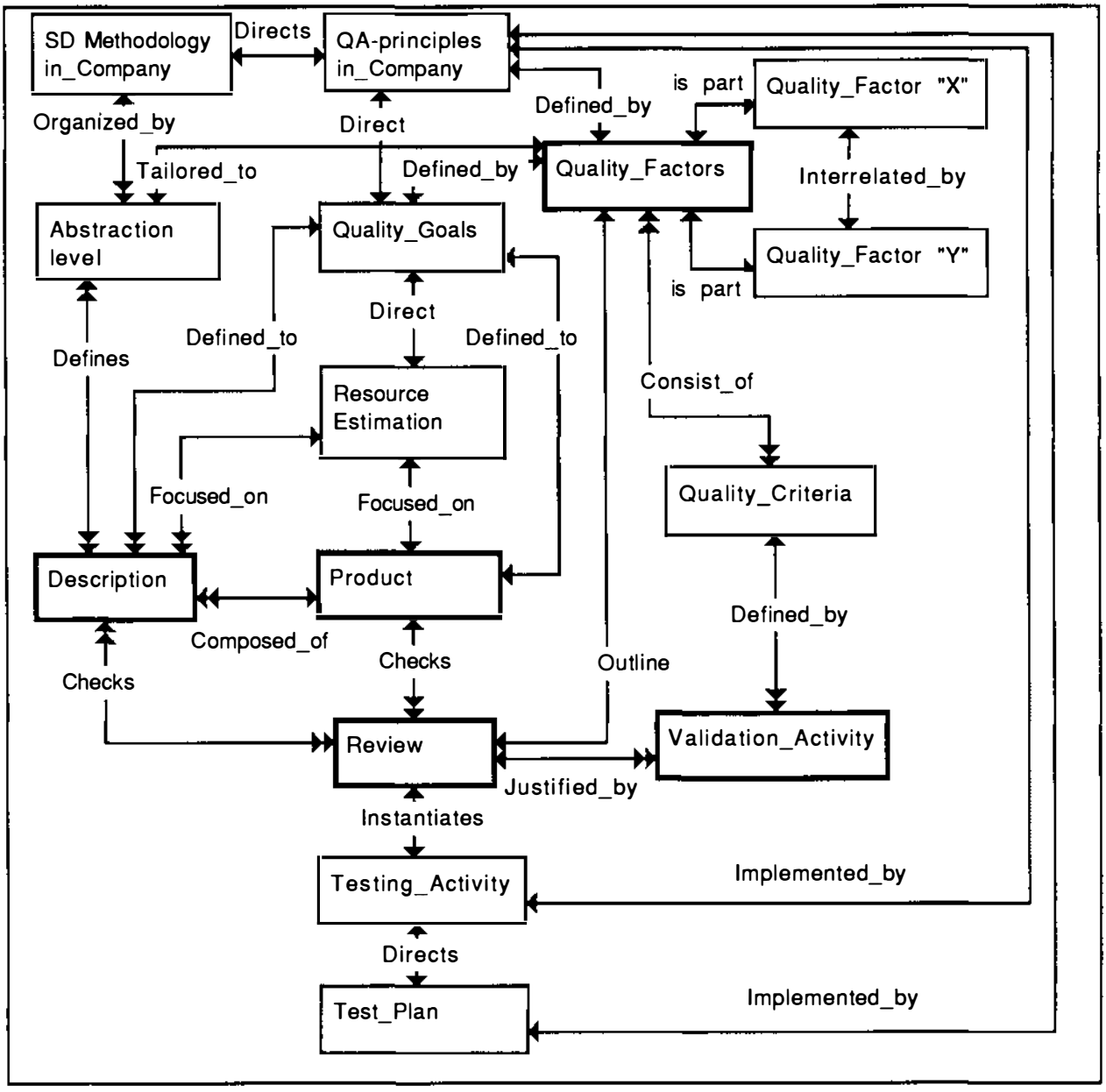


Figure 1. The main concepts in Quality Assurance

In addition to the levels of abstraction the QA world defines three aspects which we consider and visualize in the Quality Game (c.f. Figure 2), namely quality factor interrelationships and refinements (Aspect 2), the resource and quality relationship (Aspect 3) and software validation (Aspect 4).

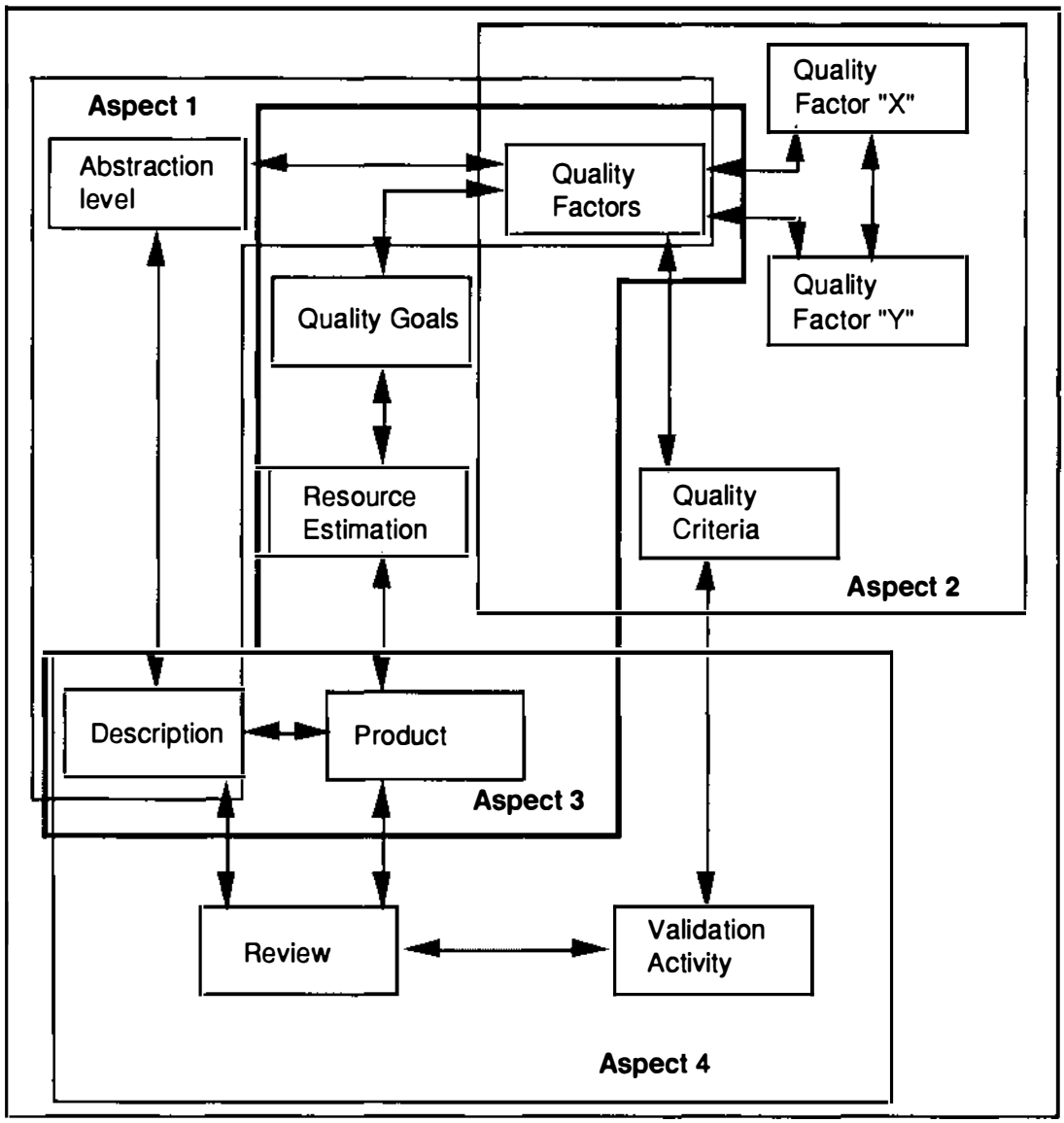


Figure 2. Four aspects of the Quality Game

The bold line (Aspect 3) depicts the final goal of the Quality Game, the resource estimation for a specific product. This is based on quality values (estimated) for alternative design and code descriptions, and efforts (estimated) consumed in producing these descriptions.

Due to the complexity of the quality issue, it is not possible to define step by step the path by which a novice can become an expert in this area. The ladder in Figure 3 is intended to illustrate the possible layers, how one could reach the professional level in quality issues, and how the Quality Game especially supports this maturing process.

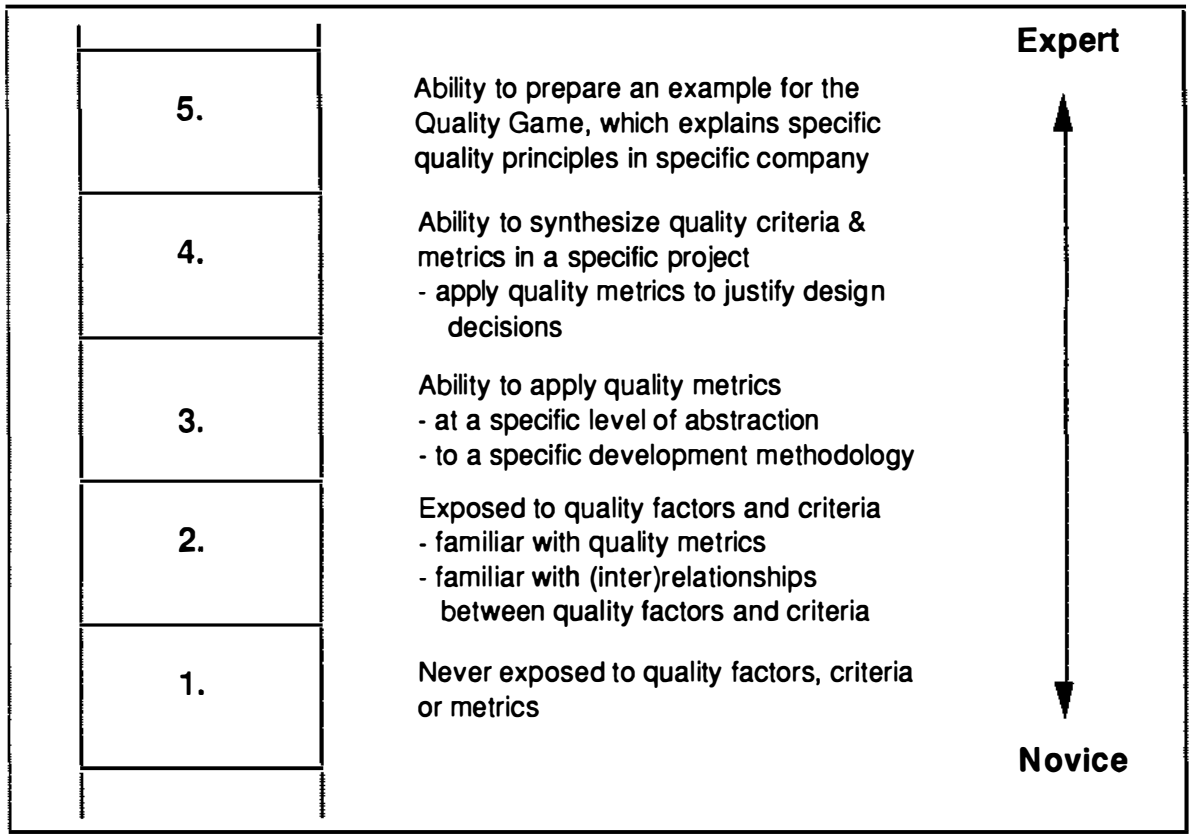


Figure 3. The ladder for becoming a quality issue professional

The path from novice to expert in the quality field, especially by means of the Quality Game, in which criteria and checklists are tailored to an object-oriented development methodology, is similar to learning object-oriented designing and programming, i.e. learning is based on a reading - writing cycle. In the Quality Game it means that we first read existing prepared examples and finally prepare our own one.

3. MAIN WINDOWS OF THE QUALITY GAME

The four aspects will be presented using a separate window for each (except aspect 4 will be considered in two windows). This chapter defines briefly the background to each window. We will then consider and walk through a prepared example in detail in the next chapter.

The main windows of the Quality Game and their relations to the four aspects are presented in Figure 4. The logical order for proceeding through these windows is to start from the Quality window (i.e. the quality factors and their interrelationships & refinements), and then move via the Abstraction Level (AbstrLevel) window (i.e. considering the criteria at each specific abstraction level by means of an object-oriented development methodology) to the Resource Estimation (ResEst) window, and if necessary from a justification viewpoint, shift to the Description window (i.e. to consider the criteria-driven validation and the justification for the specific description).

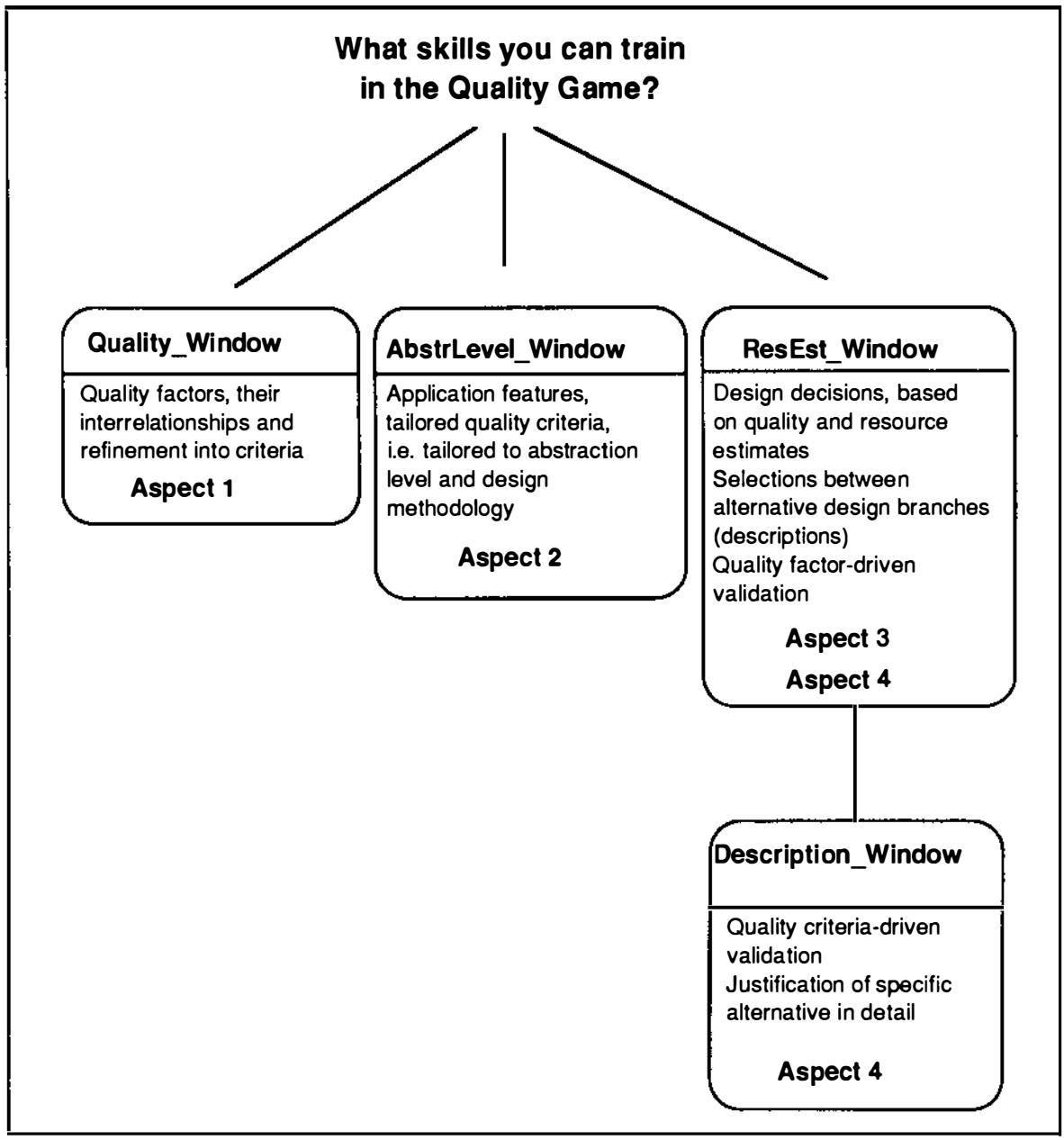


Figure 4. The training goals of the Quality Game

The Resource Estimation window forms the core of the Quality Game. It supports resource and quality relationship presentation, and combines with the Description window to present two ways for achieving quality-driven validation.

A more illustrative description of the Quality Game is presented in Figure 5, where the four windows are divided into panes. The Smalltalk environment, on top of which the Quality Game is built, supports four pane types, i.e. text, list, graph and tree panes. This is also the reason why the different viewpoints on specific aspects are implemented using these panes.

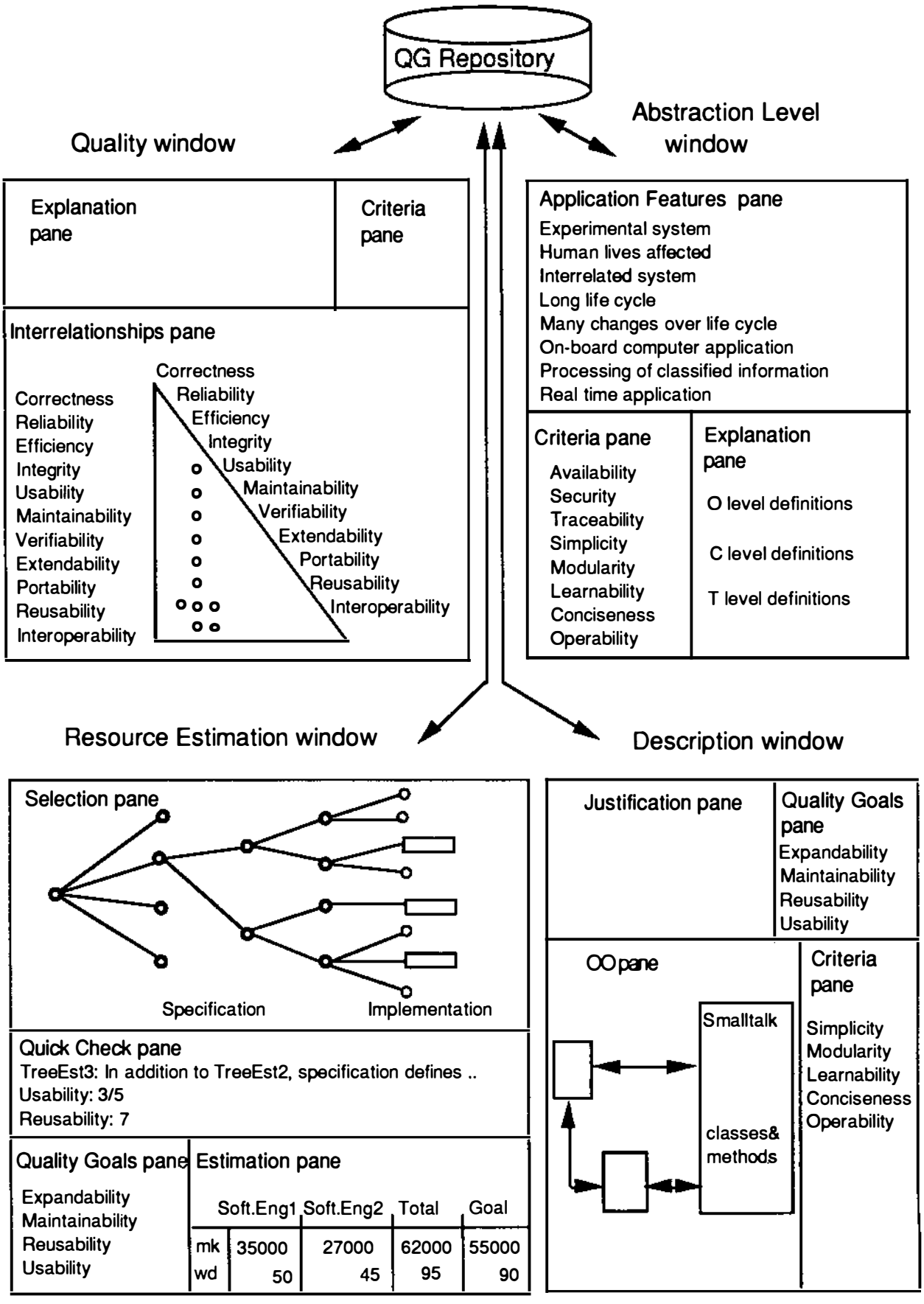


Figure 5. Outline of the Quality Game

3.1 The Quality window

The Quality window presents the interrelationships between the quality factors and their refinement into criteria (sub-factors). An understanding of potential quality factors and interrelationships between these and the means for breaking the factors down into criteria forms the basis for our further analysis.

The main approaches to considering software quality are stated by Boehm et al. (1978) and McCall et al. (1977). Some alternative approaches (e.g. Gilb 1976, 1988) and also synthesising work (the REQUEST approach e.g. Kitchenham (1987) and Petersen and Kitchenham (1988), the Quality Assurance approach in Evans and Marciniak (1987), Perry (1987) and the IEEE Standard (1989)) are then presented.

Where do the factors come from? Evans and Marciniak (1987) present three categories related to specific areas of the software life cycle, which classify the factors as follows:

Acquisition concern	User concern	Quality factor
	How well does it utilize a resource?	Efficiency
Performance	How secure is it?	Integrity
How well does it	What confidence can be placed in what it does?	Reliability
function?	How easy is it to use?	Usability
	How well does it conform to the requirements?	Correctness
Design	How easy is it to repair?	Maintainability
How valid is the design?	How easy is it to verify its performance?	Verifiability
	How easy is it to expand or upgrade its capability or performance?	Expandability
Adaptation	How easy is it to change?	Flexibility
How adaptable is it?	How easy is it to interface with another system?	Interoperability
	How easy is it to transport?	Portability
	How easy is it to convert for use in another application?	Reusability

Table 1. Software quality factors (Evans and Marciniak 1987)

This set of factors is very similar to that of the REQUEST approach, which also includes Understandability, Testability and Survivability and uses the term Extendability instead of Expandability. The Flexibility factor omitted from it. The set used in the IEEE standard is almost the same as that of Evans and Marciniak one, but as in the REQUEST set, it includes Survivability. Our set of quality factors is that of Table 1, except that we did not included Flexibility in the set due the difficulty of specifying it (c.f. Booth 1989).

We consider four kind of quality issue in the Quality window; quality factor definitions, quality factor interrelationships, refinement of factors into criteria, and criteria definitions. The definitions of quality factors are presented as follows (IEEE Standard 1989):

Reusability: A set of attributes that bear on the ability of software to be reused in applications other than the original one.

To make the factors measureable, we need metrics, which also causes a problem. We can always measure ratios, like achieved level/ planned level, but the

identification of more specific metrics is problematical. Reusability metrics is an example which is more than pure ratio metrics (Kitchenham 1987).

Reusability metrics: (effort expended to produce the component - effort expended to incorporate the component in a new system)* n where n is the number of new systems which will use the component

The second issue, interrelationships between factors, is defined by Perry (1987) in the form:

Reusability - Efficiency: The generality required for a reusable system increases overhead and decreases the efficiency of the system.
Reusability - Integrity: Generality required by reusable software provides severe protection problems.

The third issue, the set of criteria fixed to the quality factors, is presented in Table 2. The criteria and their definitions have been collected from several sources, e.g. Evans and Marciniak (1987), Gilb (1988) Perry (1987), Booth (1989) and IEEE Standard (1989).

Efficiency:	Time Economy, Resource Economy
Integrity:	Auditability, Security, Survivability
Reliability:	Non Deficiency, Error Tolerance, Availability, Accuracy
Usability:	Usefulness, Ease of Use, Learnability, Likeability, Operability
Correctness:	Completeness, Consistency, Traceability
Maintainability:	Consistency, Readability, Conciseness
Verifiability:	Simplicity, Modularity, Readability
Expandability:	Readability, Upgradeability, Conciseness
Portability:	Hardware Independence, Soft. System Independence, Installability
Reusability:	Readability, Modularity, Software Independence
Interoperability:	Modularity, Communication Commonality, Data Commonality

Table 2. Quality factors broken down into criteria

Due to our goal of applying the general criteria and checklists to an object-oriented development methodology, we have taken the stronger term Readability instead of Self-Descriptiveness. The Integrity factor is more powerful here than in its usual presentation (e.g. Evans and Marciniak 1987) including also term Survivability, which is presented as a quality factor in some contexts (e.g. Kitchenham 1987). Usability as presented here is an adaptation of Booth (1989), and emphasizes the transcendent feature of the factor, i.e. according to Kitchenham and Walker (1986) it cannot be defined precisely and is something that is felt rather than measured.

Althought it is difficult to find appropriate metrics for some of the criteria, some have numerous metric scales, e.g. Modularity can be measured in terms of module size, data coupling, span of control and module strength.

3.2 Abstraction Level window

The Abstraction Level window presents the levels of abstraction of the OCT (Organizational, Conceptual, Technical) framework (Iivari 1989) and its implications for the quality factor context. Software validation ties the abstraction levels and quality factors together, as it utilizes the major characteristic of abstraction, i.e. focus on a specific viewpoint and the making of design decisions from that viewpoint. In the OCT framework this means that decisions at the O level are based on organizational impact, i.e. what is the best alternative from the viewpoint of the whole organization, those at the C level (which consists of the sub-levels Universe of Discourse, Specification and User Interface) mostly on user satisfaction, and those at the T level (which consists of the sub-levels Architecture and Implementation) on implementation aspects such as reusability and efficiency.

At the O level we consider the software from whole utility viewpoint, i.e. we must define the potential users and their requirements. We must also select the most important quality factors which control our requirements definition. The selection process is supported by the following suggestions:

Experimental system:	Expandability
Human lives affected:	Integrity, Reliability, Correctness, Verifiability
Interrelated system:	Interoperability
Long life cycle:	Maintainability, Expandability
Many changes over life cycle:	Reusability, Expandability
On-board computer application:	Efficiency, Reliability, Correctness, Integrity
Proc. of classified information:	Integrity
Real time application:	Efficiency, Reliability, Correctness

Table 3. Quality factor suggestions (adapted from IEEE Standard 1989)

These suggestions do not include the Usability factor, because it is expected to be defined for each application. Efficiency is of the same kind (c.f. Kitchenham 1987), but it is included in the suggestions to emphasize its role in real-time application.

The factors and criteria have different interpretations at different levels of abstraction, and the process/product viewpoint also causes variation. High Reusability at the C level, for example, emphasizes the process viewpoint and means that we can reuse existing solutions (e.g. we prefer reusing the Smalltalk panes in the Quality Game definitions to building our own ones). At the T level the Modularity criterion refines Reusability using the metrics module size, data coupling, span of control and module strength (c.f. Card and Glass 1990), allowing us to measure the Modularity of our solution (i.e. product). In the object-oriented design and programming case the span of control seems to be the most interesting measure for Modularity. So we measure the span of control value of the method, i.e. the number of the methods invoked by a given method. Other criteria, Readability for example, can be defined as follows

Readability: The degree to which the specific method is calling other methods in other subclasses.
The degree to which inheritance and information hiding are followed in the design of the software.

Certain factors are more relevant than others at a given level of abstraction. Integrity, Portability and Interoperability are factors which are defined in the context of the environment or organization utilizing the software, and thus they are relevant at the O level. Usability is relevant at the C level (User Interface sublevel), and Expandability, Modifiability and Reusability, for example, at the T level. This taxonomy of factors relevant at specific levels is only a suggestion and not a rule, so that same factors and especially criteria (c.f. Table 2) can be considered at different levels. The checklists fixed to these criteria are tailored to the levels of abstraction and explain the meaning of the criteria at the level.

3.3 Resource Estimation window

Resource estimation is based on the settings of goals or objectives and the player's (designer's) attempt to reach them. It bears some similarities to the Design by Objectives (DbO) principle (Gilb and Krzanik 1987), which is a project management method that supports the supervising of a project according to fixed quality objectives. The Resource Estimation window is connected with the other four windows in the manner depicted in Figure 6.

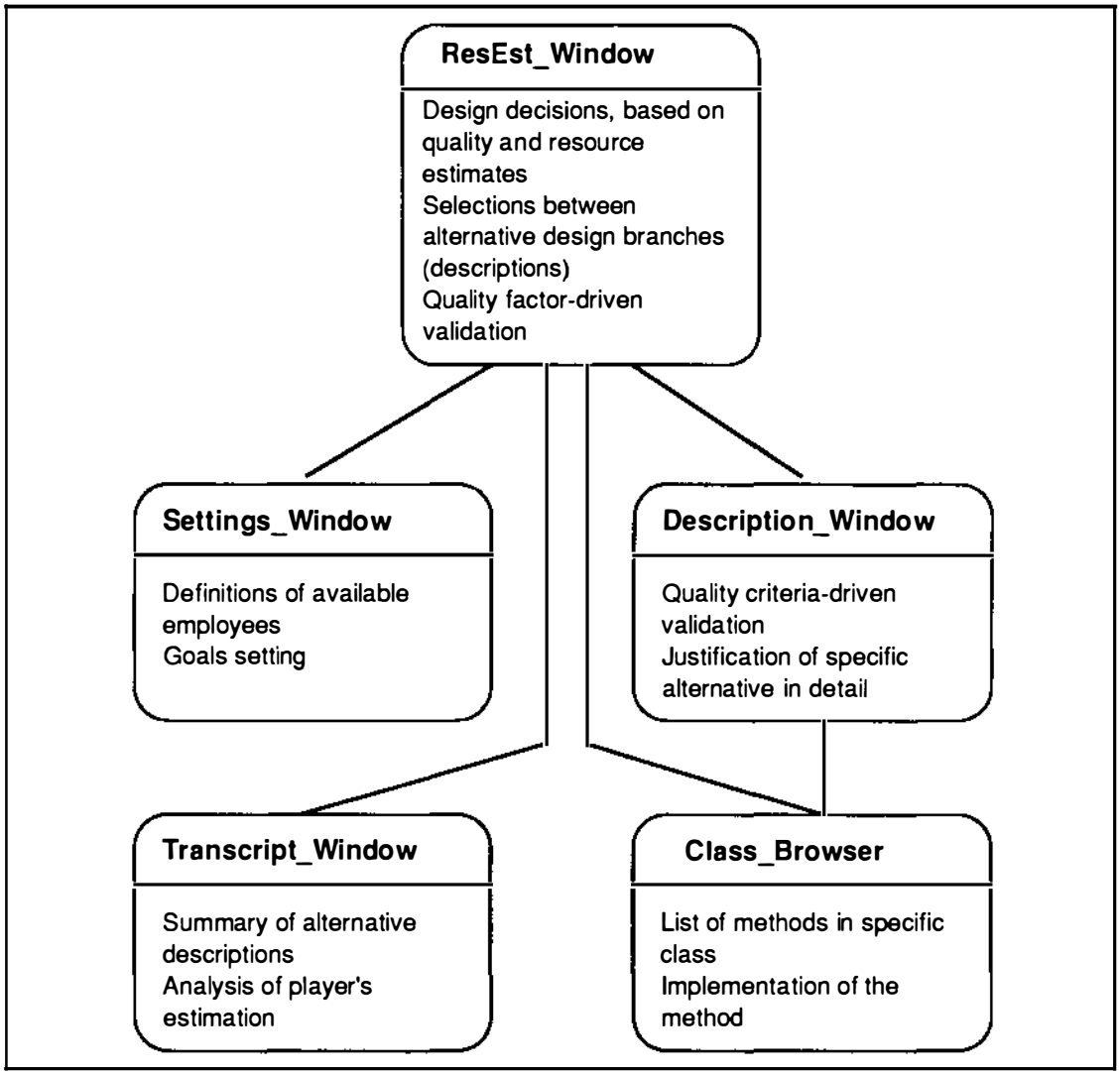


Figure 6. Connections of the Resource Estimation window

The Resource Estimation window attempts to visualize the relations between quality and resources in a specific prepared project and to assign them quality values at a rough level. The player (designer) navigates in the problem space described by the tree structure. The nodes of the tree correspond to descriptions at the C and T levels and the player decides which branch to proceed along according to the definitions, quality estimates and resource estimates contained in the descriptions.

The connected windows support the start of the game and provide for the necessary decisions. Thus we decide in the Settings window what employees will be used in the game and set the objectives for finance and throughput time. In the Transcript window we can assess alternative descriptions of the specific decision situation and the final result of our estimation process. The Description window allows us to consider the specific description in detail, i.e. why it is better than some other alternative. For justification purposes we can use existing checklists of specific criteria and add relevant justifications to the description when preparing our own example.

3.4 The Description window

The description to be assessed here and considered in more detail is selected in the Resource Estimation window, i.e. quality criteria are used to outline checklists which are defined in the Abstraction level window or tailored for this specific description of the prepared example. The Description window refines decisions made either at the C level (specifications and user interface issues) or at the T level (architecture issues). Decisions at the C level are usually justified in terms of the usability aspect and those at the T level in terms of the reusability, expandability and maintainability aspects.

The specification and user interface-based decisions are justified by plain text in the Justification pane, whereas the architecture level description can be presented by a graphics editor in the OO pane. The graphics editor understands the symbol subclass, method and dictionary and permits the usual editor activities such as move, connect, disconnect and remove. The purpose of this window is to inspect the descriptions according to the pre-reviewing principle (c.f. Tervonen 1988) The more detailed subclasses of the architecture level, i.e. methods under that subclass, can be checked in the Class browser window.

Due to the fact that software quality is difficult to capture and define, this validation aspect is very valuable. It helps us to understand the path from objectives to decisions at the implementation level, i.e. how one selection serves a specific quality goal better than another. It also directs the attention of the player (designer) to some specific detail of the example.

4. IMPLEMENTATION OF THE QUALITY GAME

Since the Quality Game is implemented using Smalltalk/V in a Macintosh environment, it was quite natural, according to the Model-View-Controller (MVC) triad (c.f. Krasner and Pope 1988) or Model-Pane-Dispatcher (MPD) triad (c.f. LaLonde and Pugh 1989), for the specific aspects to be visualized to be implemented in windows of their own. In Smalltalk/V we use the MPD triad and apply it in such a way that the different windows, i.e. Quality, Resource Estimation, Abstraction level and Description, are submodels of Quality Assurance. Each of the

windows is divided into panes (c.f. Figure 5), which present specific characteristics or relationships of the submodel.

The major part of the work of constructing the Quality Game is focused on pane definition and implementation. Smalltalk/V can utilize existing Pane classes and subclasses such as Graph pane, List pane, Text pane and Tree pane. The QG_Repository (c.f. Figure 5) is a collection of Dictionaries (Smalltalk's class for presenting collections), which also transfer information between windows.

Due to its training and teaching feature, and because we can teach only structures and principles, we must emphasize usability, reusability, expandability and modifiability in the construction of the Quality Game. In terms of Smalltalk development, this means that at the specification level, for example, we must know the existing Pane classes in order to understand the reusability status of the development. This knowledge helps us to define the tentative class hierarchy of the application (including methods), and these definitions are further implemented on top of the abstract machine (i.e. Smalltalk/V) at the architecture and implementation levels.

5. WALKING THROUGH THE QUALITY GAME IN A PREPARED EXAMPLE

The Quality window is totally example independent, explaining general interrelationships between quality factors, relationships between factors and criteria, and defining the factors and criteria. In the Abstraction Level window we can consider all criteria checklists tailored to a specific level of abstraction and object-oriented development methodology or the prepared example defines this set The Resource Estimation and Description windows consider the characteristics of the prepared example and allow both quality factor-driven and quality criteria-driven validation of the descriptions.

5.1 Quality factors interrelationships and refinement into criteria

Depending on the level of knowledge of the player, he/she can obtain information about the general definition of the specific factor, interrelationships between factors, refinement of the factor into criteria, and general definition of the specific criteria.

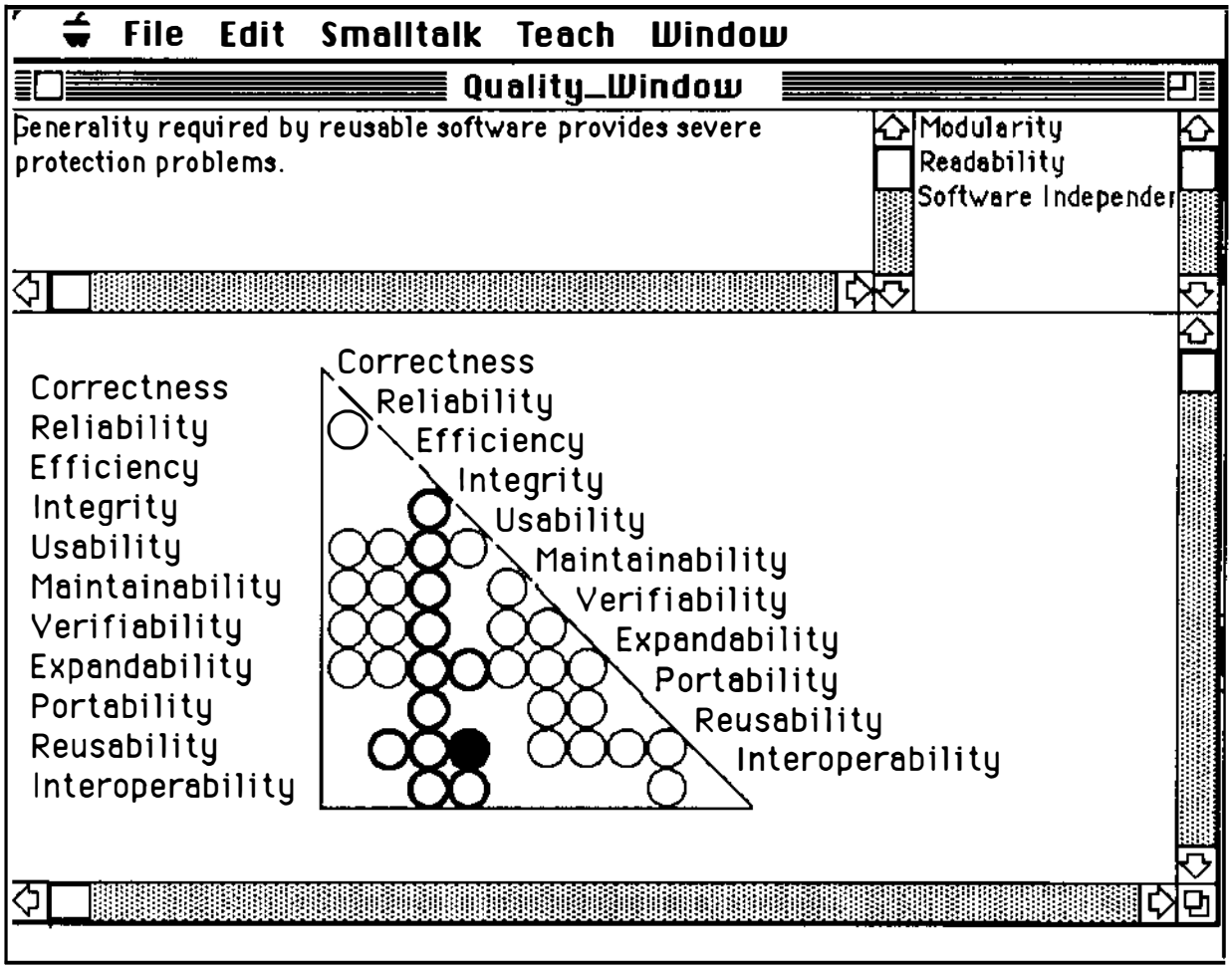


Figure 7. The Quality window

All information acquisitions are started by activating a circle in the Interrelations pane (black circle in Figure 7). According to Perry (1987) these plain circles indicate synergetic relations between two factors, while the bold circles indicate conflicts. The impact of conflicting features is that the cost of implementation will increase, which will lower the benefit-to-cost-ratio. The factor in the same line as the activated circle defines the set of criteria to be explained in the Criteria pane. The type of explanation is selected from the Teach menu.

5.2 Abstraction level perspective on quality issues

The Abstraction Level window explains the main connections between the quality issues and the levels of the OCT framework. The level to be interviewed is selected from the main menu. At the Organizational level we define the most important factors to be assessed in the example using the template, as presented in Table 3, while at the Conceptual and Technical levels the checklists of the specific criteria are tailored to these levels and to the object-oriented methodology (c.f Figure 8).

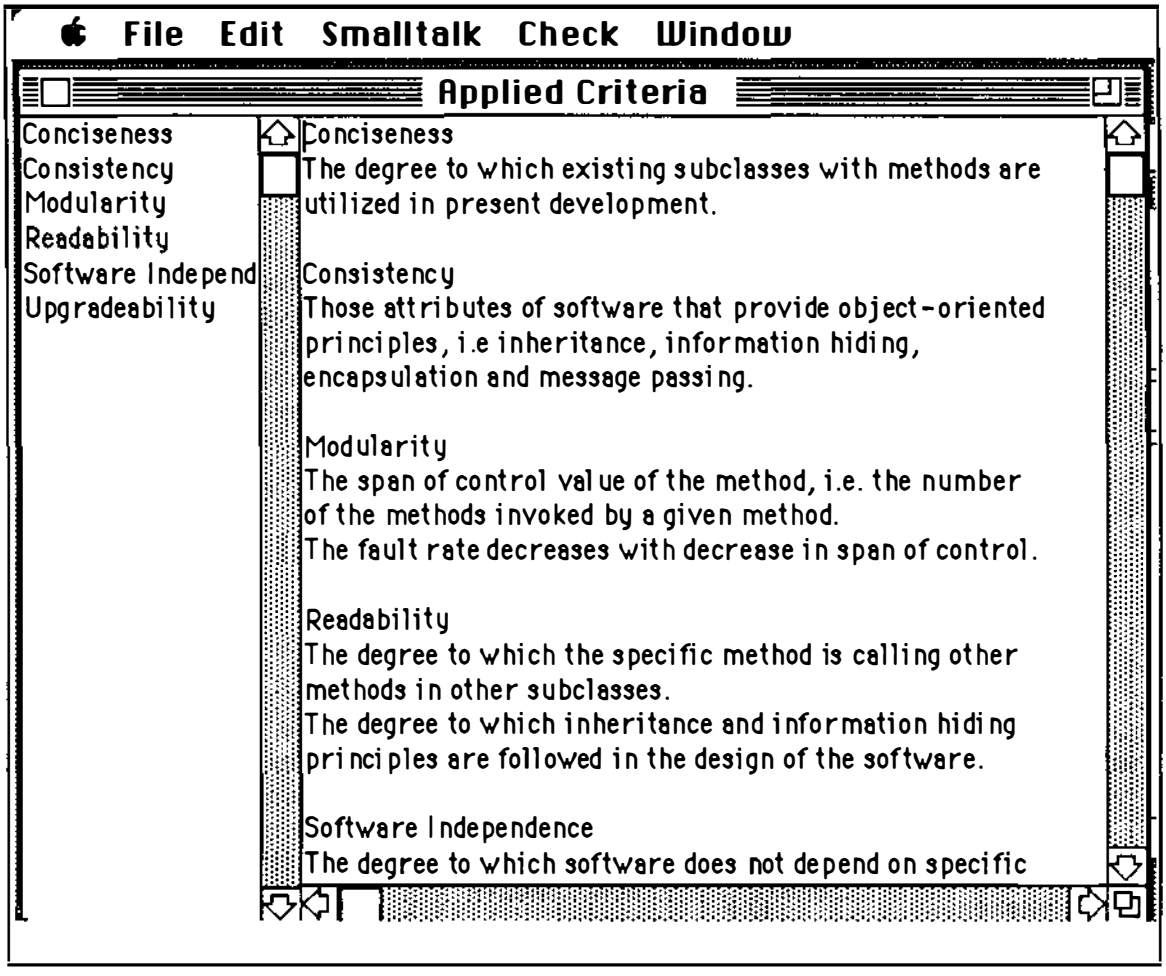


Figure 8. Applied criteria window

The set of criteria in Figure 8 corresponds to the factors Reusability, Expandability and Maintainability, which are defined to be the most important factors at the T level.

5.3 Quality and resources interrelationship

As a prepared example we use the C and T level descriptions of the Quality Game. To emphasize the decision-making situations at different levels, we have defined some intermediate levels and also some alternative branches. The starting point and the methods implemented are still the same as in the original Quality Game. The tree in Figure 9 depicts the assembly structure of the Quality Game, which means that it does not present the inheritance of the nodes.

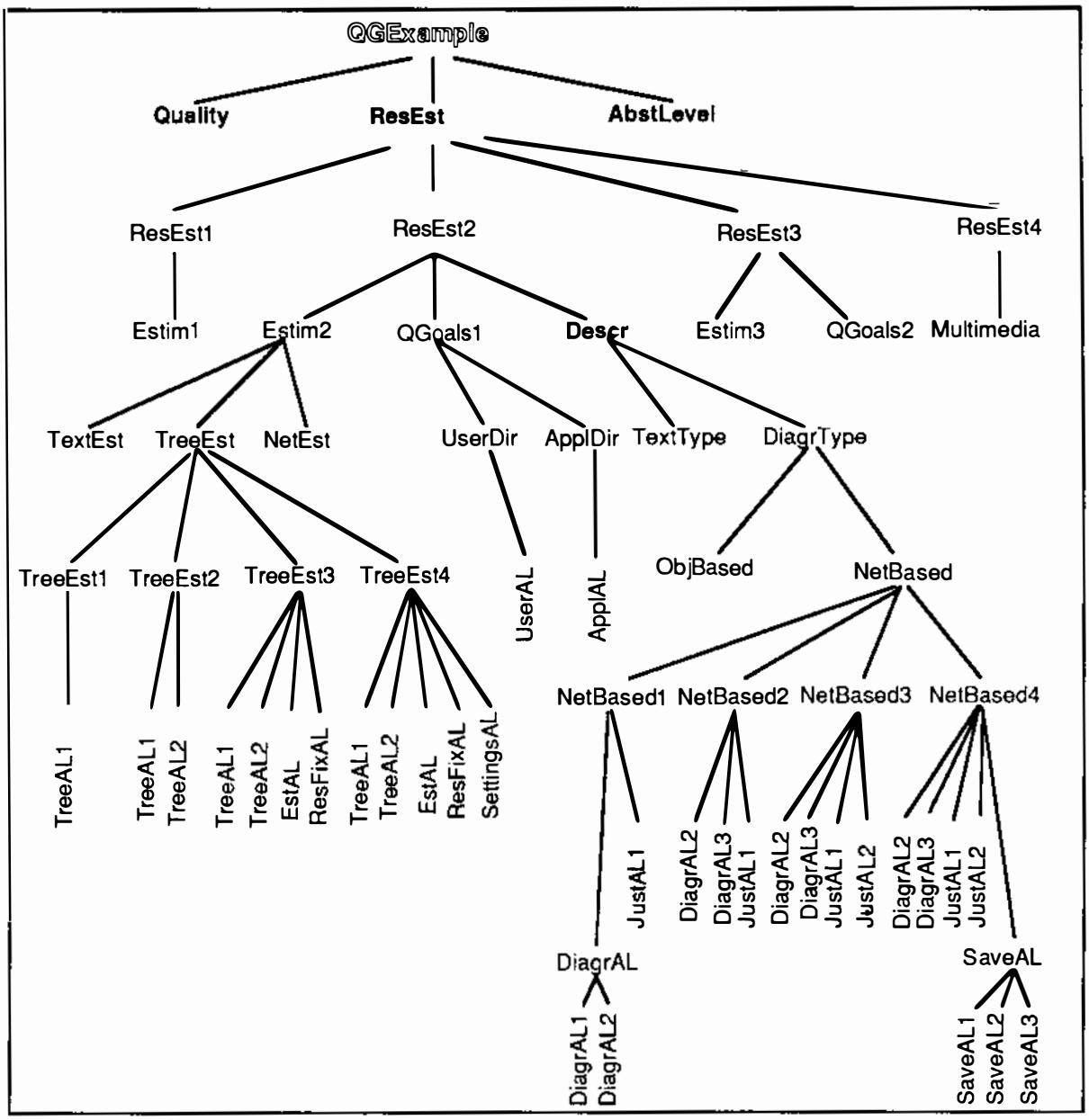


Figure 9. Tree structure of the prepared example

The relationship between quality and resources forms one aspect of project management issues. The assessing of this relationship is implemented in the Quality Game by driving the player to walk through a path from definition of the example to the leaves which present its implementation in Smalltalk methods. In each node is a situation in which the player must decide what path to follow and who is to be the software engineer for this node. We also call this path selection quality factor-driven validation, because it assesses the value of the node using estimated quality values.

At the beginning of the game the player must assign the software engineers to be utilized in producing the software. The basic set of employees comprises one from each group, where the groups are

top expert:	Smalltalk experience 2 years design and programming experience 5 years
new expert:	Smalltalk experience 6 months design and programming experience 2 years
young designer:	Smalltalk experience 3 months design and programming experience 1 year
novice:	Smalltalk experience 1 month design and programming experience 6 months

Table 4. Groups of available software engineers

The player can reserve extra workers at fixed prices: top expert 10.000 mk, new expert 5.000 mk, young designer 2.000 mk and novice 1.000 mk.

The objectives are also fixed at the beginning of the game. The player can set goals in terms of both the amount of money needed to implement the software and the time taken up by the software production. The throughput time (workdays) is the sum (real work, i.e estimated time/software engineer/node + delay time) on the longest path from definition to implementation (e.g. from node ResEst to node DiagrAL1 in Figure 9). The longest path is composed by assuming that we must wait at node TreeEst in the example tree (Figure 9) until nodes ResEst2 and Estim2 are completed but can start the nodes QGoals1 and Descr in parallel with Estim2.

The producing time / node for each group of software engineers is estimated, and the player assigns the software engineers according to this value. The general rule is that the top expert is best choice if available, but the game incorporates a nuisance feature which assumes that the top experts and new experts are also working in other projects, giving rise to random announcements that these employees are now busy and the player must either accept the delay or select some other employee to do this job. The amount of money used in the game is a sum of factors of time/employee and expenses/employee/workday. The estimated expenses in one year are: top expert 200.000 mk, new expert 170.000 mk, young designer 150.000 mk and novice 100.000 mk.

A typical decision situation is presented in Figure 10. The player must decide what branch to select. As seen from the tree structure, the alternatives are TreeEst1, TreeEst2, TreeEst3 and TreeEst4. Activation of TreeEst3 allows the short definition of the node and estimated resources in separate panes to be checked.

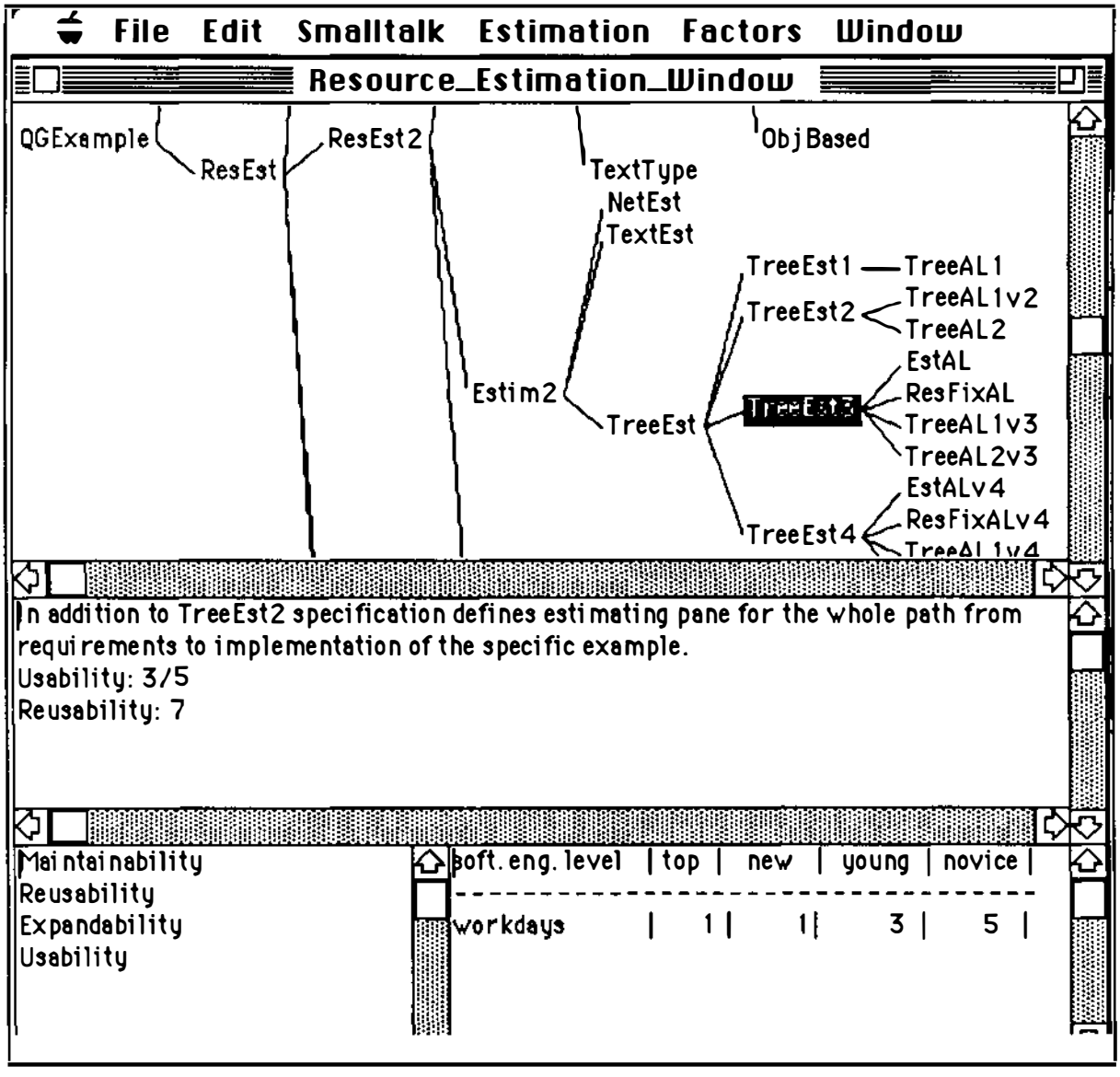


Figure 10. The Resource Estimation window

A more informative sheet is available in the Transcript window, where the alternatives are easier to compare:

TreeEst1
Specification defines a pane for the tree structure of the prepared example and a short definition of the activated node in the text pane.
Usability: 1/5 Reusability: 8
Resources: top 1, new 1, young 2, novice 4

TreeEst2
In addition to TreeEst1, specification defines a window for assessing alternative descriptions and a pane for checking estimated resources for node production.
Usability: 2/5 Reusability: 8
Resources: top 1, new 1, young 2, novice 3

TreeEst3
In addition to TreeEst2, specification defines estimation pane for the whole path from requirements to implementation of the specific example.
Usability: 3/5 Reusability: 7
Resources: top 1, new 1, young 3, novice 5

TreeEst4
In addition to TreeEst3, specification defines real game characteristics for the Quality Game, i.e. goal setting, short analysis of the solution.
Usability: 5/5 Reusability: 5
Resources: top 2, new 2, young 4, novice 6

Table 5. Alternative nodes in a decision situation

The quality factors used here are Usability and Reusability, as the TreeEst nodes are descriptions at the C level (i.e. Specification and User Interface levels). Usability employs cumulative metrics such that the ResEst node in Figure 9 has a value 10 and ResEst2 can reach values 1-9 depending on the branch selected. The nodes explained in Table 5 represent a decision situation in which the maximum value that can be reached at this level is 5. Thus the best choice from a usability viewpoint is TreeEst4. Reusability values have a scale 1-10, and the value indicates workload of the node. The values 10-7 indicate many reusable parts in design and coding, values 6-5 indicate some reusable parts and values lower than 5 should warn us of very hard work ahead in designing and programming.

The quality definitions at the T level (i.e. Architecture and Implementation levels) need factors which break down into criteria and checklists and are available in the Description window.

The final result of the resource estimation process will be inserted into the Estimation pane in the form:

	Paavo	Aino	Teuvo	Heikki	Eila	Total/ Throughput	Goal/ Throughput
mk	30.000	40.000	25.000	35.000	20.000	150.000	130.000
workdays	50	80	75	50	40	120	100

Table 6. Final estimation form

In this case the player has selected one extra software engineer and has accepted the extra expense. The throughput time is the longest path from definition to

implementation. A more detailed analysis of the final estimation is available in the Transcript window, where any missed nodes and poor solutions will be indicated.

5.4 Validation of descriptions

The Description window refines the node assessment started in the Resource Estimation window. Quality estimation was factor-driven in that window, whereas it is criteria-driven in the Description window, which means that we break the factors down into a set of criteria and assess the quality of the description from that viewpoint. The justification of a specific alternative description is presented as a checklist and linked to given criteria. The player can walk through existing justifications by activating specific criteria in the list. In the example (Figure 9) there are two nodes (DiagrAL and StoreAL) for alternative implementations, although the benefits of other leaves can also be justified by Description windows. The Figure 11 below is an example of validation by justification at the T level (i.e. quality goals are defined using factors Expandability, Reusability and Maintainability).

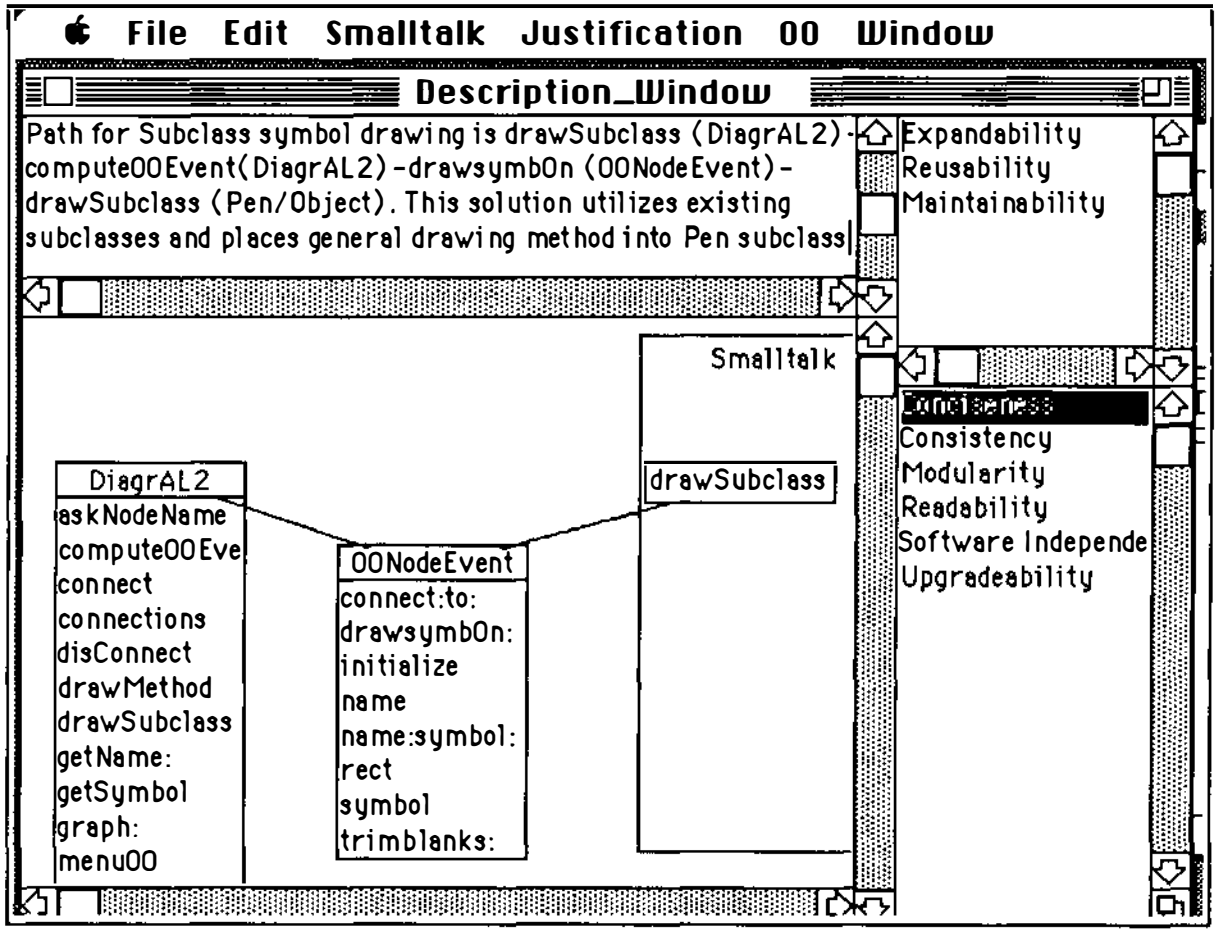


Figure 11. The Description window

The rationale in the Justification pane is a statement applied to this case. It is possible to accept any text in this pane as a checklist and add to the specific criteria. The graphics editor supports object-oriented description drawing, i.e. it understands the drawing symbol subclass, method and dictionary, and accepts the

functions move, connect, disconnect and remove. This window can also be used when assessing the nodes at the C level from the criteria viewpoint (using factors Usability and Reusability). If necessary, the subclasses with methods can be checked in detail in own Class Browser window.

In Figure 11 the general checklist of Conciseness is refined by statement, which justifies the contents of DiagrAL2 subclass. In this case the the method drawSubclass is implemented utilizing existing subclasses as much as possible. The final method, which draws the symbol, is placed into Pen subclass. This solution is not necessary best from Readability viewpoint, but we justify it here from Conciseness viewpoint.

6. CONCLUSIONS

Visualization of abstract issues such as Quality Assurance is difficult. Our experiment with delimiting a microworld and playing the Quality Game in it is one approach. The present version forms a core for the Quality Game and implementation solutions (i.e. Smalltalk/V) allow high expandability.

Although only one window in the present Quality Game has the characteristics of a real game, we hope this gives some visions of advanced versions of games. The minor improvement could be more effectiveness in searching the best path, for example. Other interesting and challenging directions for further development could be based on development of some tools for modularity and complexity measurement in an object-oriented development methodology and multimedia approach (e.g. company presentation, Quality Assurance culture presentation using multimedia tools).

There are two principles which must be retained in future versions of the game: support for company-specific training and support for a reading-writing cycle in training. Company specific training means that because the Quality Assurance culture of each company is unique, it is necessary to prepare a separate example for each. The reading-writing cycle in training means that we first learn from prepared examples and then write our own example.

7. ACKNOWLEDGEMENTS

The work was supported by Tauno Tönning Foundation. The author wishes to acknowledge Juhani Iivari and Lech Krzanik for interesting discussions during the construction of the Quality Game, and referees for their comments and suggestions.

8. REFERENCES

Boehm B.W., Brown T.R., Kasper H., Lipow M., Macleod G.J. and Merritt M.J, 1978, Characteristics of Software Quality, North-Holland

Booth P., 1989, An Introduction to Human-Computer Interaction, Lawrence Erlbaum Associates Ltd

Card D.N., and Glass R.L., 1990, Measuring Software Design Quality, Prentice Hall

Clancey W.J., 1987, Intelligent tutoring systems: A tutorial survey, in: van Lamsweerde A., and Dufour P., (eds) Current Issues in Expert Systems, Academic Press

Gilb T., 1976, Software Metrics, Studentlitteratur

Gilb T., 1988, Principles of Software Engineering Management, Addison-Wesley

Gilb T., and Krzanik L., 1987, Design by Objectives, ACM SIGSOFT Software Engineering Notes, vol 12, no 2

IEEE Standard, 1989, IEEE Standard for a Software Quality Metrics Methodology (Draft), IEEE Inc.

Iivari J., 1989, Levels of Abstractions as a Conceptual Framework for an Information System, in: Falkenberg E.D. and Lindgreen P. (eds), Proc. IFIP WG 8.1 Working Conference on Information System Concepts: An In-depth Analysis, Namur, Belgium

Kitchenham B.A., and Walker J.G, 1986, The meaning of quality, in Barnes D, and Brown P. (eds.), Software Engineering 86, Peter Peregrinus Ltd

Kitchenham B.A., 1987 Towards a Constructive Quality Model, Software Engineering Journal, vol 22, no 4

Krasner G.E., and Pope S.T., 1988, A Cookbook for Using the Model/View/ Controller User Interface Paradigm in Smalltalk-80, Journal of Object-Oriented Programming vol 1, no 3

LaLonde W., and Pugh J., 1989, Pluggable Tiling Windows, Journal of Object-Oriented Programming, vol 2, no 3

McCall J.A, Richards P.K, and Walters G.F., 1977, Factors in Software Quality, Volumes I, II, and III, RADC reports

Papert S., 1980, Mindstorms, Basic Books

Perry W.P., 1987, Effective Methods of EDP Quality Assurance, in: Schulmeyer G.G., and McManus J.I., (eds), Handbook of Software Quality Assurance, Van Nostrand Reinhold Company

Petersen P.G., and Kitchenham B.A., 1988, The Development of a Software Quality Model, in Proceedings of the First European Seminar on Software Quality, Brussels, Belgium

Shu N.C., 1988, Visual Programming, Van Nostrand Reinhold Company

Tervonen I., 1988, Validation by Pre-reviewing and Justification, in Proceedings of the 6th PNSQ Conference, Portland

Ward P.T., and Mellor S.J., 1985, Structured Development for Real-Time Systems, Volume I: Introduction & Tools, Yourdon Press, New York

Prototype Testing Tools†

Andrew E. Babbitt
Shari Tims Powell

Department of Computer Science
Portland State University
Box 751
Portland, OR 97207

Abstract

Testing tools are software analyzers that use information from particular executions of a program as well as information about the program text itself. Research prototypes of such tools are essential to investigate the ideas they embody. Often, hand calculation is so tedious and error-prone that an investigator cannot obtain any intuition about his or her ideas without an implementation to aid in experiments. Traditionally, such tools have been implemented in conventional high-level languages (e.g., C, Pascal), a process that takes more time than a prototype should. The technology of compiler generators and logic programming, applied to the idea of self-instrumenting programs, promises to drastically shorten the prototype cycle.

This paper describes the general method for implementing a prototype testing tool, gives an example of an existing system fitted into the method, and discusses the ease with which such prototypes may be changed.

Keywords: instrumented programs, test coverage, testing tools

Biographies

Andy Babbitt is a recent graduate of the Portland State University Computer Science program.

Shari Powell is working on her BS in Computer Science at Portland State.

They are both members of a National Science Foundation research group working on program testing at Portland State.

† Work supported by NSF grant CCR-8822869 under the direction of Professor Richard Hamlet.

Prototype Testing Tools

Andrew E. Babbitt
Shari Tims Powell

Department of Computer Science
Portland State University

1. Introduction

Research prototypes of program-testing tools are often essential in developing testing methods. Systematic testing involves not only many test cases, but information from the program specification, information about the program structure, and even details of the execution history. In a typical method, the bookkeeping tasks are so extensive that hand simulation is literally impossible, and to gain understanding of the method's strengths and weaknesses requires experiments with a working test tool. Unfortunately, the difficulty of understanding a new method means that such a tool will need to change as the experiments suggest modifications.

It is common practice to implement research prototypes in a conventional programming language like Pascal (cf. ASSET [Weyuker], STAD [Laski]). No matter how "quick and dirty" the implementation is, the prototype takes months to write, and is as hard to modify as any medium-sized program. We suggest a different method for prototype tool construction, based on three established computing technologies:

(1) *Self-instrumented programs*. Instead of monitoring test executions of the program under an interpreter, it is usually possible to outfit the program with monitoring statements in its own language, interspersed with the original program statements in such a way that when the program executes, the added statements collect relevant information to perform the necessary analysis. This technique has been in use for at least 15 years [Stucki].

(2) *Table-driven parser generators*. Compiler compilers are widely available that construct a parser for a programming language from the language grammar. (The UNIX system compiler compiler is a combination of the tools called *lex* [Lesk] and *yacc* [Johnson].) Compiler compilers usually allow some form of context-sensitive syntax-directed translation, so that as the constructed parser identifies the input source program, arbitrary actions can be performed. In a compiler, these actions build a symbol table; when the parser is used to create self-instrumented programs, they add the instrumentation statements.

(3) *A logic-programming language*. Languages like PROLOG have several important advantages in creating prototype analyzers. PROLOG can be used to analyze facts about programs and executions expressed in a database fashion, and can be used to query those facts interactively. The declarative PROLOG programming style lends itself to describing software-analysis such as testing methods. [Harrison] has used PROLOG in this way for a static FORTRAN analyzer.

We have combined these technologies to achieve very rapid generation of program-analysis tools, which are easy to change. The time from conception to running prototype should be an order of magnitude less than for conventional development in a high-level language. Changes in the language analyzed by the tool, and in the analysis algorithms, should be trivial to implement.

In brief, what we propose for the analysis paradigm is the following:

(a) The program to be analyzed is parsed using a parser automatically generated from a grammar. During the parse, syntax-directed translation techniques are used to write a collection of PROLOG facts, expressing the information needed for later static analysis.

(b) Also during the parse and using syntax-directed translation, self-instrumentation is generated for the program being analyzed. The instrumentation takes the form of statements that when executed at run-time, generate PROLOG facts about the behavior of the program.

(c) The program is executed on test data, and the combined facts of (a) and (b) form a PROLOG database describing program and test.

(d) PROLOG rules added to the database describe the analysis to be performed on the collected facts of (c).

(e) The user interface to the analysis system is the PROLOG query mechanism, in which the library predicates of (d) are an aid to investigation of the database of (c).

2. Introduction to PROLOG

Logic programming, particularly in the widely-available language PROLOG [Clocksin], is ideal for writing software-analysis programs. Because PROLOG may be relatively unfamiliar to those who usually work with imperative languages, we will give a brief introduction to the language and a description of the features we use.

2.1. Facts

Rather than specifying a particular algorithm to solve a problem, programmers in PROLOG specify known facts and relationships about a problem. A fact itself is simply the relationship between two or more arguments. An argument can be a constant, a variable, a nested structure or a list. Variables in PROLOG begin with an upper case letter; constants begin with lower case. A classic example is planning routes through a maze given facts about the possible paths from one room to another. Suppose the facts for the maze were:

```
path(a,b).
path(b,c).
path(c,d).
path(b,d).
```

In the first fact above, `path` is the name of the fact and a and b are its arguments. This fact may be read "there is a path from room a to room b." In this simple example we can quickly see from the facts that there are two possible routes from room a to room d: (a,c,b,d) or (a,b,d).

2.2. Rules

We can specify a PROLOG rule which will determine whether or not there is a path from one room to another. A PROLOG rule is composed of a head and a body (optional) separated by a `:-`. The body of the rule puts conditions on the head allowing PROLOG to determine what values the

variables in the head can have which make the rule true. A rule for determining if a route is possible given the path from room to room is:

```
route(X,X).
route(X,Y) :- path(X,Z), route(Z,Y).
```

The first line above is called a base case: `route(X,X)`, which says that there is a path from a room to itself. This particular rule needs a base case since it is recursive. The rule on the second line contains two conditions separated by a comma which is the PROLOG equivalent of logical AND. The second line would therefore be read as: "there is a route from X to Y IF there is a path from X to Z AND a route from Z to Y."

2.3. Queries

The set of facts and the rule above comprise the PROLOG database in this introductory example. To the user, facts and rules are indistinguishable, therefore they are referred to collectively as predicates. This database can be queried using the name of a predicate. For example, at the PROLOG prompt (?-) we can ask if there is a route from room b to room d:

```
?- route(b,d).
```

to which PROLOG will respond

```
yes
```

To answer the query, PROLOG searched the database for a match. We can also include variables in our query and let PROLOG tell us what possible values the variable may have. For example the query:

```
?- route(b,X).
```

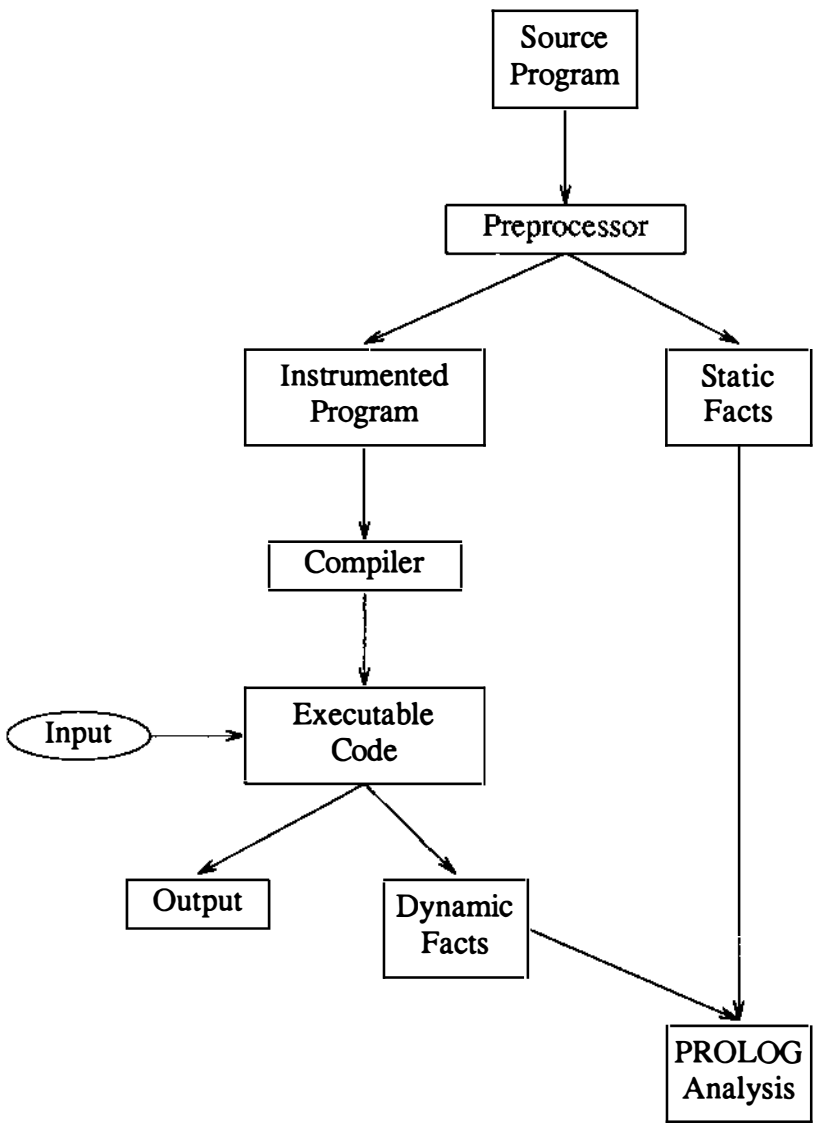
where X is a variable (since it begins with an upper case letter), will result in the response:

```
X = b ;
X = c ;
X = d
yes
```

The semicolons in the above example were typed in by the user; a semicolon is the equivalent of logical OR. The result above may then be interpreted as: "there is a possible route from b to b OR b to c OR b to d." When the user typed the semicolon, PROLOG continued searching the database (where it left off) to find any additional X values for which the predicate `route` was true.

3. Explanation of the Method

The parser-generator takes a grammar for a subset of C and produces a parser which, given a C source program, produces an instrumented program along with a set of static facts. In this section a simple program is used to demonstrate this testing method using branch coverage [Howden]. Static facts revealing possible paths the program could take and dynamic facts showing the path actually taken for a particular data set are shown. A portion of the instrumented version of the program is given which shows what statements are inserted into the original program in order to generate the run-time facts. Finally, sample PROLOG predicates are used to analyze branch coverage and demonstrate the query feature of the testing tool. The following diagram shows an overview of the entire process:



3.1 Triangle Program

The following is a simple program which identifies types of triangles given the length of their sides, assumed to be in decreasing order [Ramamoorthy]. Line numbers have been added to the program for reference convenience.

```
main  ( )
{    int  a,  b,  c,  d;

1    fscanf( stdin, "%d %d %d" , &a, &b, &c)   ;

2    while ( a != -1  )
     {
3         if ( a >= b  &&  b >= c )
          {
4              if ( a== b  ||  b== c )
               {
5                   if ( a == b  &&  a == c )
                    {
6                        printf( "equilateral\n" )   ;
                    }
7                   else {    printf( "isosceles\n" )   ;
                             }
               }
               else  {
8                   a*= a  ;
9                   b*= b  ;
10                  c*= c  ;
11                  d= b+ c  ;

12                  if ( a != d )
                    {
13                       if ( a < d )
                         {
14                            printf("acute scalene\n");
                         }
15                       else   {  printf("obtuse scalene\n");
                                  }
                    }
16                  else {  printf("right scalene\n");
                            }
               }
          }
17        else  { printf( "illegal\n" )   ;
                  }
18        fscanf( stdin, "%d %d %d" , &a, &b, &c)   ;
     }
19   printf( "end of triangle program\n" )   ;
}
```

3.2. Static Facts

The parser itself generates static facts about the program. The following facts, pertaining to the possible steps (**pstep**) the program could take, are produced when the above program is parsed:

```
pstep(1,2).
pstep(2,3).
pstep(3,4).
pstep(4,5).
pstep(5,6).
pstep(5,7).
pstep(4,8).
pstep(8,9).
pstep(9,10).
pstep(10,11).
pstep(11,12).
pstep(12,13).
pstep(13,14).
pstep(13,15).
pstep(12,16).
pstep(3,17).
pstep(17,18).
pstep(16,18).
pstep(15,18).
pstep(14,18).
pstep(7,18).
pstep(6,18).
pstep(17,18).
pstep(18,2).
pstep(2,19).
```

These facts indicate that a possible step can be taken from one statement to the next. For example, **pstep(5,6)** and **pstep(5,7)** indicate that either statement 6 or statement 7 may follow statement 5, i.e. a branch occurs at statement 5.

3.3. Instrumented Program Segment

Following is a segment of the instrumented version of the program. The `fprint(fileptr...)` statements in lighter type are the instrumentation added to produce run-time facts. (The parser added declarations at the beginning of the program for the variables `xstmt` and `fileptr`). Normally, the instrumented statements are separated by semicolons and added following the line they pertain to. Exceptions to this occur when adding the statements to an **if**, **for**, or **while** statement. In the example above, the instrumentation statements are added to the program's **if** statement by separating the expression of the **if** by logical **&&** (and) operators. The instrumentation statements must be inserted prior to the if expression because otherwise the short-circuit evaluation of relational expressions performed by the C compiler might cause them

to be missed. Line numbers are for the original program.

```
5                    if ( (!fprintf(fileptr,"step(%d,5).\n",xstmt))
                          && (xstmt = 5) &&  (a== b&& a== c) )
                     {
6                         printf( "equilateral\n" )   ;
                          fprintf(fileptr,"step(%d,6).\n",xstmt) ;
                          xstmt = 6 ;
                     }
                     else
                     {
7                         printf( "isosceles\n" )   ;
                          fprintf(fileptr,"step(%d,7).\n",xstmt) ;
                          xstmt = 7 ;
                     }
```

The first fact in the example above is generated as part of the **if** statement execution, namely, **step(4,5)**, which expresses that statement 5 was executed after statement 4. At each statement such a fact is generated (at statement 6 the fact **step(5,6)** is produced, or at statement 7 **step(5,7)** and so on), thereby creating a run-time trace of the order in which the statements were executed.

3.4. Run-time Facts

When the instrumented program executes, the dynamic (**step**) facts are generated. Given the following input data file (which includes one of each of the types of triangles the program can identify):

```
1 1 1
3 2 2
5 4 3
6 5 4
6 5 3
-1 0 0
```

the following **step** facts (among others) are generated (only those facts of interest to the forthcoming discussion are listed, i.e., facts generated at a branch statement):

```
step(2,3).
step(3,4).
step(4,5).
step(5,6).
step(5,7).
step(4,8).
step(12,16).
step(12,13).
```

```
step(13,14).
step(13,15).
step(2,19).
```

If one were to analyze the facts by hand, the possible steps the program could have taken at each branch statement could be compared to the dynamic **step** facts above. The following static facts, generated by the parser, describe branch statements:

```
pstep(2,3).
pstep(3,4).
pstep(4,5).
pstep(5,6).
pstep(5,7).
pstep(4,8).
pstep(12,13).
pstep(13,14).
pstep(13,15).
pstep(12,16).
pstep(3,17).
pstep(2,19).
```

Hand analysis of the **step** and **pstep** facts reveals that the test data failed to take the step (3,17). The **step(3,17)** was absent because the test data did not take the else branch in statement 3 (to statement 17). The next section describes how PROLOG predicates can be used to perform this branch analysis.

3.5. PROLOG Queries

PROLOG rules can be written to use the static and dynamic facts generated at parse-time and run-time respectively. For example, a PROLOG rule such as the following can be written to indicate which branches were not taken:

```
branch_not_taken(X,Y) :- pstep(X,Y), pstep(X,Z), Y \== Z,
                         not step(X,Y).

branches_not_taken(S) :- setof([X,Y],X^branch_not_taken(X,Y),S) ;
                         S = [].
```

The first rule above, **branch_not_taken** can be read as: "There is a possible step from X to Y AND a possible step from X to Z AND Y is not equal to Z, AND the program when executing did not take a step from X to Y. The second rule, **branches_not_taken** uses the predefined PROLOG predicate **setof** to find all the branches which were not taken.

Given the database of **step** and **pstep** facts in section 3.4, one could make the following query:

```
?- branches_not_taken(BNT).
```

with the result:

```
BNT = [ [3,17] ]
```

The results of the query indicate that new data needs to be added to the input file to cover the missing branch. A new data point is added and the instrumented program is executed using the following input file:

```
1 2 3
1 1 1
3 2 2
5 4 3
6 5 4
6 5 3
-1 0 0
```

The new run-time file contains the following **step** facts occurring at branch statements:

```
step(2,3).
step(3,17).
step(3,4).
step(4,5).
step(5,6).
step(5,7).
step(4,8).
step(12,16).
step(12,13).
step(13,14).
step(13,15).
step(2,19).
```

Now when making the same query,

```
?- branches_not_taken(BNT).
```

the result is

```
BNT = []
```

that is, there are no branches which the new data set did not cover.

4. Changes to the Prototype Testing Tool

The prototype testing tool, like any system, may not be exactly what the user wants. This section will examine making two possible changes to the system: changing the language and changing the coverage method.

4.1. Changing the Language

The user might want a tool to test programs in another language. Changing the system to test Pascal programs, for example, would require three steps:

1. Alter the lex program to return Pascal tokens.
2. Obtain a Pascal grammar in yacc format.
3. Add syntax-directed actions to Pascal rules.

4.1.1. Lex and Yacc

The prototype testing tool was constructed with the aid of the UNIX tools lex and yacc. Lex writes programs that recognize examples of regular expressions. The lex user supplies regular expressions and actions to be performed when the input matches the expression. Lex generates a program that recognizes the expressions in its input and performs the specified actions. The lex-generated program supplies tokens to the yacc parser-generator.

Yacc writes a subroutine that recognizes grammar rules. The yacc user supplies the input-structure rules and actions to be performed when a rule is recognized. The yacc-generated subroutine uses the lexical analyzer and performs the actions specified by the user when a rule is recognized. These actions can be any legal C code.

Code examples are shown below so that the similarities between C and Pascal can be seen, but the reader can ignore the details.

4.1.2. C to Pascal Conversion

The first step, altering the lex program to return Pascal tokens, requires changing the regular expressions. For example, C comments are defined in lex:

```
"/*"[^\*\/]*"*/"        { ; }    /* ignore comments      */
```

The corresponding Pascal version would be:

```
"{"[^}]*"}"|"(*"[^\*\)]*"*)"      { ; }    /* ignore comments */
```

The C assignment operator is defined:

```
(=)     { strcpy(yylval.string,strcat(yytext,fin));
                        return(ASSIGNOP); }
```

The corresponding Pascal version would be:

```
(:=)     { strcpy(yylval.string,strcat(yytext,fin));
                        return(ASSIGNOP); }
```

Similar changes and additional definitions would handle the reserved words and other tokens.

The next step is building the parser with yacc. The first step in this process is to get the parser to recognize correct Pascal programs and output them. First the grammar is entered into the yacc format, and **strcpy**, **strcat** and **printf** functions are used to build up and write the output. In the if-statement for example, the only change necessary would be to not explicitly print the left and right parenthesis surrounding the expression. The following rule and actions would output a C **if** statement:

```
statement:      IF  LPAREN expr RPAREN  THEN statement  optElse
                { strcpy($$,strcat($1,strcat($2,strcat($3,
                strcat($4,strcat($5,strcat($6,$7))))));
                printf("%s\n",$$); }
```

The corresponding Pascal version would be:

```
statement:      IF  expr THEN statement  optElse
                { strcpy($$,strcat($1,strcat($2,strcat($3,
                    strcat($4,$5))));
                printf("%s\n",$$); }
```

With the parser-generator now outputting the input program, the next step is to add the instrumentation and static-fact generation. Because C and Pascal are so similar, the actions are almost the same. Because the parser-generator uses C, the rest of the action code would remain the same. Procedures called by the action code would need only minor modification; the **printf** statements would have to be changed to **writeln** statements, as the following example shows:

```
printf(" ((fprintf(fileptr,%cstep(\%%d,%d),\\n%c,xstmt))",
                        prs[0],statmnt,prs[0];
```

The corresponding Pascal version would be:

```
printf(" (((writeln(fileptr,%cstep(\%%d,%d),\\n%c,xstmt))",
                        prs[0],statmnt,prs[0];
```

The additional changes would be to those sections of code that add instrumentation to the program. In the C version, naming the dynamic facts file **states** and adding the integer **xstmt** takes only three lines.

```
Program:        Preprocessor_Directives
                    { printf("\nFILE *fileptr;\n");
                    printf("int xstmt = 0;\n"); }
                ExternalDefs   /* function definitions */
                MainProgram
            ;
```

A third line inside the **MainProgram** rule opens the file.

The Pascal version would be more complicated because of the way files are defined as variables. The following segment adds the file name to the program.

```
Program:        PROGRAM
                     { printf("%s ",$1); }
                identifier LPAREN
                     { printf("( states,",); }
                identifier_list RPAREN SEMICOLON
                block PERIOD
            ;
```

Additional actions added to the grammar rules for **const_decl**, **type_decl** and **var_decl** complete the file definition and add **xstmt** as **VAR** of type integer. The other instrumentation added to the program being tested prints the **step** facts. Because C syntax allows assignment statements and function calls inside expressions, the instrumentation is added inside loop statements to print the **step** facts. In Pascal this is not the case. In the Pascal version, instrumentation would be added prior to, and immediately following looping statements. This would result in some step facts being printed twice around for and while loops, but would not cause the PROLOG predicates any problem.

We think that the lex program to define the tokens could be rewritten for Pascal in about an hour. Given the grammar definition in yacc format, a working Pascal parser-generator could be written in about 20 hours. It would be necessary to change less than 150 lines of the current 800 lines in the C parser-generator.

Although using yacc is a considerable saving over writing a parser from scratch, parsing is often inappropriate for our application. In most cases, the detailed syntax analysis a compiler needs is an overkill. On the other hand, where we could use more power and convenience, in the syntax-directed actions, yacc is inadequate. A variant of a compiler-compiler intended to produce only preprocessed programs, not compilers, would be more appropriate. [Purtilo] describes a system which may be an improvement.

4.2. Changing Type of Coverage Tested

Dataflow coverage methods are more complex than the branch coverage of Section 3.5. One such method attempts to cover the control points that lie between a variable definition and its use. These execution chains are called def-use paths [Rapps]. If all def-use paths within a program are covered, then every possible path from a variable's assignment to its use has been executed. To determine def-use path coverage, two additional static facts are required:

```
assigned_in(VarName,LineNo)
used_in(VarName,LineNo)
```

To determine if a def-use path was not covered, we will compare the possible def-use paths with the def-use paths actually executed, using the PROLOG predicates:

```
pdupath(VarName,Begin,End,PDUP)
xdupath(VarName,Begin,End,XDUP)
```

To get the new facts into the static database, actions must be added to the parser-generator. Specifically, actions are required where an assignment statement is recognized, and where an expression uses a variable to write the facts.

To generate the **used_in** fact it is necessary to add to the static database file every time a variable is used in an expression. In the grammar we used, this fact can be added by adding a **printf** statement into the grammar's actions in the two places an identifier appears in an expression.

To generate the **assigned_in** fact is not much more complicated. The parser-generator calls the procedure **insert** as an action whenever a statement has been recognized. This procedure inserts the instrumentation into the program being tested. By adding code that determines the type of statement recognized, we can write the **assigned_in** fact for every assignment statement.

With these changes, the portion of the static-facts file generated by lines 1-3 of the triangle program (cf. Section 3.2) is:

```
assigned_in(a,1).
assigned_in(b,1).
assigned_in(c,1).
used_in(a,2).
pstep(1,2).
used_in(a,3).
used_in(b,3).
used_in(b,3).
used_in(c,3).
pstep(2,3).
```

These static facts are adequate to calculate the potential def-use paths in a program. The essential rules are:

```
path(X,X,[X]).
path(Beg,End,[Beg|T]) :- pstep(Beg,Mid), path(Mid,End,T).
```

which compute paths as lists of statement numbers.

The dynamic facts (using **step**) described in Section 3.4 cannot similarly be used to generate paths which are actually executed because they contain no information about the order in which the steps were executed. For a program with backward branching (e.g. the WHILE loop in the triangle program), the **step** facts can indicate false paths that arise from different choices on separate passes through the loop. One way to filter these out is to add a sequence counter to **step**, which is incremented each time a **step** fact is written to the dynamics facts file, so that the dynamic facts for the test data

```
3 3 1
3 3 3
```

become:

```
step(0,0,1).
step(1,1,2).
step(2,2,3).
step(3,3,4).
step(4,4,5).   (*)
step(5,5,7).
step(6,7,18).
step(7,18,2).
step(8,2,3).
step(9,3,4).
step(10,4,5).
step(11,5,6).  (**)
step(12,6,18).
```

The initial sequence number can be used to eliminate the potential false path from (*) to (**). The change required to modify production of the **step** fact in the parser generator is trivial.

To make these changes to our system required adding 23 lines of code to the parser-generator and took about three hours.

The complete PROLOG rules to calculate actual def-use paths are:

```
dxpath(C,V,X,X,[X]).
dxpath(Count,Var,First,End,[First|T]):- step(Count,First,Second),
                        not(assigned_in(Var,First)),
                        NextCount is Count + 1,
                        dxpath(NextCount,Var,Second,End,T).

xdupath(Var,Begin,End,[Begin|T]) :-     assigned_in(Var,Begin),
                        used_in(Var,End),
                        step(C,Begin,Second),
                        dxpath(Count,Var,Second,End,T).
```

The rules for the possible def-use paths are similar, but use **pstep** instead of **step**.

Finally, the predicate to return unexecuted def-use paths is:

```
notx_dupath(Var,T) :-          pdupath(Var,Begin,End,T),
                        not(xdupath(Var,Begin,End,T)).
```

Using the triangle program of Section 3 and the data:

```
3 2 1
6 5 4
1 2 3
-1 0 0
```

will illustrate the new analysis abilities of the system. To find all the possible def-use paths for the variable **a** that begin at statement 5:

```
?- pdupath(a,Begin,5,Path).
```

```
Begin = 1
Path = [1,2,3,4,5] ■

Begin = 18
Path = [18,2,3,4,5]
```

To find all the executed def-use paths for the variable **a** that begin at statement 8:

```
?- xdupath(a,8,End,Path).

End = 12
Path = [8,9,10,11,12] ;

End = 13
Path = [8,9,10,11,12,13]
```

The query for def-use paths not covered is:

```
?- notx_dupath(a,DUPX).

DUPX = [1,2,3,4,5] ;

DUPX = [18,2,3,4,5] ;
```

From this we can see that the program never checks the condition **a == b && a ▬ c** in line 5, and that more data is needed to cover this def-use path.

5. Summary

The combination of technologies we have described achieves rapid generation of program-analysis tools, which are easy to change. Language independence arises naturally in the parser generator, whose driving grammar may be easily changed. The syntax-directed translations change only slightly for different languages, and the use of PROLOG to record static facts minimizes the difficulty of change. One might even hope that *all* relevant information could be collected, so that the syntax-directed part of the parser is fixed for all languages.

Similarly, one might hope to collect all relevant facts about executions, but failing that, changes to the generated parser to collect new facts are small, and easy to make.

Much information of interest for program analysis can be obtained directly from the static and run-time facts generated by the system. However, should significantly different analysis be required, performing it in PROLOG is a large saving over conventional-language implementation. There are no data structures to change. It is easy to add predicates to the system to simplify the analysis.

In this paper we have used a very simple example program to demonstrate our prototyping method. The tools created of course apply in more complex situations, and we intend to investigate new testing methods, and experiment with our tools on a variety of programs.

Acknowledgement

This work was performed under the direction of Professor Richard Hamlet whose ideas and guidance made this project possible.

References

[Harrison]

W. Harrison, PDSS: a programmer's decision support system, *Data & Knowledge Engineering* 4 (1989), 115-123.

[Howden]

W. Howden, Methodology for the generation of program test data, *IEEE Trans. on Computers* C-24 (May, 1975), 554-559.

[Johnson]

S. C. Johnson, Yacc: yet another compiler compiler, *UNIX Programmer's Manual, v. 2,* Holt, Rinehart and Winston, 1983, 353-387.

[Laski]

J. Laski, Data flow testing in STAD, *J. Systems Software* 12 (1990), 3-14.

[Lesk]

M. E. Lesk and E. Schmidt, Lex - a lexical analyzer generator, *UNIX Programmer's Manual, v. 2,* Holt, Rinehart and Winston, 1983, 388-400.

[Purtilo]

J. M. Purtilo and E. L. White, Using program adaptation techniques for test coverage in Ada programs, to appear in the Eighth Annual Pacific Northwest Software Quality Conference, Portland, OR, October, 1990.

[Rapps]

S. Rapps and E. Weyuker, Selecting software test data using data flow information, *IEEE Trans. Software Eng.* SE-11 (April, 1985), 367-375.

[Stucki]

L. G. Stucki, Automatic generation of self-metric software, *Proceedings IEEE Symposium on Computer Software Reliability,* New York, 1973.

[Weyuker]

P. G. Frankl and E. J. Weyuker, A data flow testing tool, *Proceedings SoftFairII, Software Development Tools, Techniques, and Alternatives,* San Francisco, 1987, 46-53.

USING PROGRAM ADAPTATION TECHNIQUES FOR ADA COVERAGE TESTING

James M. Purtilo Elizabeth L. White

Computer Science Department
University of Maryland
College Park, Maryland 20742

ABSTRACT

During the course of software development, program metric tools are often employed to analyze the software. Examples include static program analysis, code profiles, and coverage analysis. Automated tools can facilitate this study by collecting data and generating the desired measures. Unfortunately, the implementation of such tools has proven to be difficult and time consuming.

In order to simplify the task of constructing analysis tools for Ada, we have examined a new parser generator system (NewYacc) for use in a source to source transformation approach. Our goal is to allow developers to easily and compactly express what information they need from the input language in order to compute the desired statistics. One result of our research is AINT, an instrumenter and tester for Ada source code which was constructed using the new translation technology. AINT measures both statement and branch coverage of Ada programs, allowing users to evaluate the adequacy of their test data. This paper summarizes NewYacc, and then describes the AINT implementation.

James Purtilo is a member of the Computer Science faculty at the University of Maryland, where he also holds an appointment in the UM Institute for Advanced Computer Studies. Dr. Purtilo's research deals with distributed programming and software engineering support environments. Dr. Purtilo received his PhD in Computer Science from the University of Illinois at Urbana in 1986; prior to that he received a BA and MA in Mathematics from Hiram College and Kent State University, respectively, both in Ohio. He is a member of ACM and IEEE Computer Society. Email: purtilo@cs.umd.edu

Elizabeth White is a graduate student in Computer Science at the University of Maryland, working on transformational approaches to program visualization and analysis. Previously Ms White taught computer science at Dickinson College, and received her MS and BS from the College of William and Mary in Virginia. Ms White is a member of ACM. Email: lizw@cs.umd.edu

1 OVERVIEW

Software metrics are important to understanding the software development process. In order to employ such techniques, the software being studied must either be analyzed manually, or some automatic tool must be used to process the programs. While small programs may be reasonably processed by hand, this is too time consuming for analysis of larger software systems. Automated tools are necessary, but historically these tools have proven complex to implement.

The complexity in development of automated tools is the result of several factors. When the metric requires static analysis of programs, the implementor must have access to parsing and other intermediate program representations which are usually not available for general use from a vendor. Vendors market a custom analysis system and users are not able to take this software and modify it for their purposes. Because of this, tools developed for static analysis must be built from scratch or assembled from other previously built pieces.

Other metrics require the more dynamic approach of installing code into the programs to perform the metric computation. Previously, implementors have constructed tools for profiling, coverage, and other metrics by using a low-level approach for adding code as it is compiled. In order to build these tools, the developer needs to know not only about parsing and symbol information, but aliasing, preserving coherent data representation and code generation as well.

Many of the static and dynamic metrics can be defined in terms of the language structure as defined by grammars. This suggests an alternative to the low-level approach traditionally used for program analysis. The idea is to adapt source programs at a high level so that existing compilers, code generators and other processing tools can be employed without alteration. In order to develop automatic tools for use at this high level, a technology which integrates the parsing process with the computation of metrics is needed. Users need the ability to do high-level source to source translations quickly and easily by specifying what needs to be done in terms of the language constructs themselves.

Our current research involves such a translation tool called NewYacc. NewYacc allows users to associate actions and parse tree traversals with the grammar rules for a language. Use of this tool is simple and straightforward. Starting with a grammar for the language being analyzed, actions consisting of outputs, function calls, and tree traversals are added. The resulting parser does source to source translations, either computing the desired metrics directly, or inserting source code for later dynamic measurement.

In order to evaluate this approach, our work so far has been on two different analysis tools, one static and the other dynamic. ASAPP is an Ada Static Analyzer Program which computes several different metrics including Halstead and cyclic complexity. ASAPP works by traversing the parse tree for the input program and making function calls at different points in the tree to record information for the necessary analysis. The second tool, AINT, is the focus of this paper. AINT is an Ada branch and statement coverage tool. Given an input program, AINT modifies the program in a way that allows a tester to run the output program multiple times on different data sets, and while recording which statements and branches were executed on each data set. By knowing that their test data has exercised most of the execution paths, implementors can have greater confidence that design and implementation errors in the software have been exposed, resulting in increased program quality.

In the next section we introduce the NewYacc development tool, including simple examples. In Section 3, an overview of the AINT testing system is given along with an example of how we augment Ada programs. Finally, we give a short description of how AINT was developed using NewYacc.

2 NEWYACC

Our approach to solving the problems with the generation of automatic tools to operate on complex languages is to use a translation mechanism which allows high-level source to source translation. An example of this kind of mechanism is NewYacc, an extension of the Unix compiler generator tool Yacc [John79]. Yacc takes as input a set of grammar rules with associated actions and produces a parser for the language defined by the input grammar. The parser generated by YACC parses an input stream bottom-up, discarding the parse tree as it reduces by the rules of the grammar. Actions specified by the user are performed as the reductions occur. When the parsing is done, the parse program terminates.

In NewYacc, a parser is also produced from an input set of grammar rules and actions; however, in this parser the parse tree is retained after an input stream is processed. When the sentence has been accepted, the NewYacc system allows the tree to be traversed and additional actions to be performed in user-defined ways. The actions are rewrite rules associated with the productions in the language.

A rule translation in NewYacc is of the form:

```
[(translation_label) translation_body]
```

where the translation_label is a list of translation labels and the translation_body is a list of grammar symbols, strings, statements, and user-defined functions. A traversal in NewYacc is a user-specified dynamic walk through the parse tree and corresponds to a single translation of the input. The translation labels and grammar symbols control the path of the traversal. At each internal node, the translation items are evaluated in order and at each leaf node, the contents are simply output.

For example, a grammar for expressions using only + and − as operators can be written in infix notation as follows:

```
Expression ■ Expression '+' Expression |
             Expression '-' Expression |
             digit
```

A parse tree for the input sentence "3 + 2 − 4" is shown in Figure 1. If the task to be performed is the transformation of infix expressions to either postfix expressions or prefix expressions, this could be specified in NewYacc with the following actions:

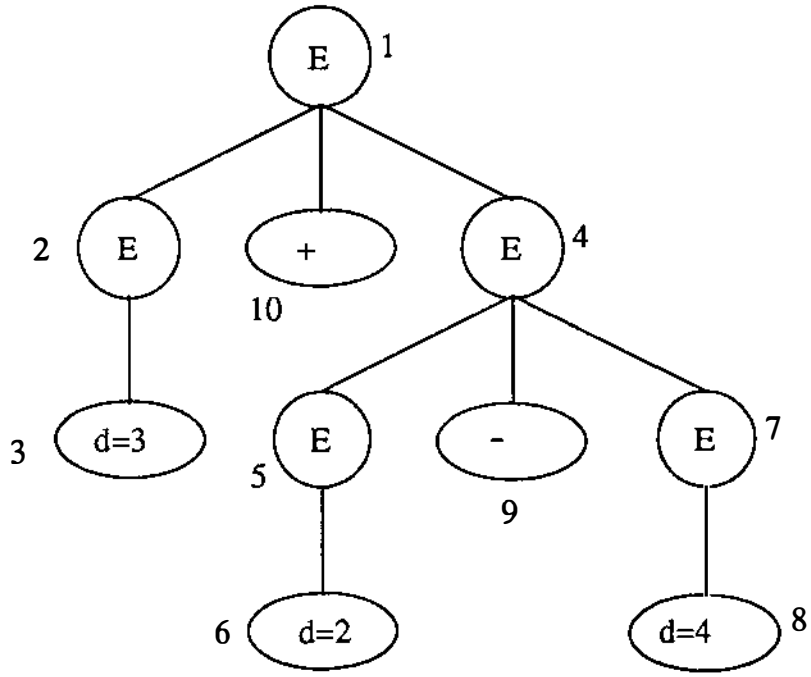


Figure 1: Parse Tree for $3 + 2 - 4$.

```
Expression    ::=   Expression '+' Expression
                    [(PREFIX) #2 " " #1 " " #3 ]
                    [(POSTFIX) #1 " " #3 " " #2]          |
                    Expression '-' Expression
                    [(PREFIX) #2 " " #1 " " #3 ]
                    [(POSTFIX) #1 " " #3 " " #2]          |
                    digit
```

The "#" indicates the traversal of the specified subtree, where the subtrees correspond to the different elements on the right hand side of the grammar rule and are numbered from left to right starting with 1. In the above example, the "#1" in the first NewYacc action specifies that the subtree for the first **Expression** is to be traversed. Literal strings such as " " are output when encountered in the translation body. Above, the first NewYacc action for the POSTFIX traversal indicates a POSTFIX traversal of the first subtree, output of the literal string " ", traversal of the third subtree as POSTFIX, output of the string " ", and finally traversal of the second subtree. If a postorder traversal was desired then the nodes of the parse tree of Figure 1 would be visited in the order indicated by the numbering and the output would be "$324 - +$" simply by specifying a traversal using the POSTFIX translation label. Use of the PREFIX label would cause the output to be "$+3 - 24$".

The above example shows an open traversal on the parse tree; other features of NewYacc include conditional traversals, use of scope variables, and selective traversals in which leaf nodes are not output. For a more complete language description, see [PuCa89].

3 AINT

The purpose of this section is to describe the AINT tool itself, including a concrete example to illustrate its application. As mentioned earlier, the development of AINT demonstrates how easily individual users can devise their own desired transformations on a source language using the NewYacc system. However, details concerning just *how* NewYacc can generate an AINT tool will be covered in the next section.

Statement coverage is the computation of which statements in the input program were actually executed during a test run. Branch coverage involves examination of all places where control flow can be divided, and informing the tester which execution branches were not followed. This information is necessary for developers to judge the adequacy of their testing data: if they find their test data has caused all parts of the application to be exercised, then they would presumably have increased confidence that any potential faults have been realized; in turn, the more faults that can be recognized through testing, the greater is the likelihood that program errors will be exposed.

In program instrumentation, code is added to an application program so that each time a significant event occurs during a run of the program, the event is recorded for later analysis. Any program testing system of this type has several responsibilities: statically, it must provide a language processor to install this instrumentation code; at run time it must record the execution of statements and branches; and after execution it must analyze the execution trace and inform the user about statement and branch coverage.

The AINT system to instrument code fulfills all of these responsibilities. Each Ada package to be instrumented is input to a tool that outputs a new version of the code (suitably augmented to containing instrumentation calls), a pretty-printed version of the source, and the list of testing obligations to be fulfilled. Subsequently, the obligation lists from all the Ada packages are concatenated and run through a tool which creates a new Ada package (containing the obligations tables for statement and branch coverage); this package of data structures is compiled and linked in with the instrumented packages, along with a driver module. The result is an executable program that not only functions as the original application, but also keeps track of all testing obligations as they are fulfilled. A diagram of the AINT system architecture is in Figure 2.

Once the test program has been created, it can be run any number of times to test sets of data. For example, if the Towers of Hanoi program shown in Figure 3 was executed with an input greater than 0, all of the statements in the main program and in procedure **hanoi** would be executed. On an input of 0, none of the code inside the conditional statement in procedure **hanoi** would be executed although the rest of the code would execute. When AINT is used to instrument the Towers of Hanoi, this new version can be compiled and run, producing the output shown in Figure 4.

In the first run, all of the statements and branches were executed, but in the second run, the input 0 causes the true branch of the conditional and all of the enclosed statements to be output as uncovered after the run. The statement numbers in the output correspond the the statement locations in a pretty printed version of Hanoi.

The input source code needs to be augmented in such a way that the output version is functionally equivalent to the original code except for the added code to compute coverage. To do this, coverage information needs to be computed before the execution of every line. The initial version of AINT

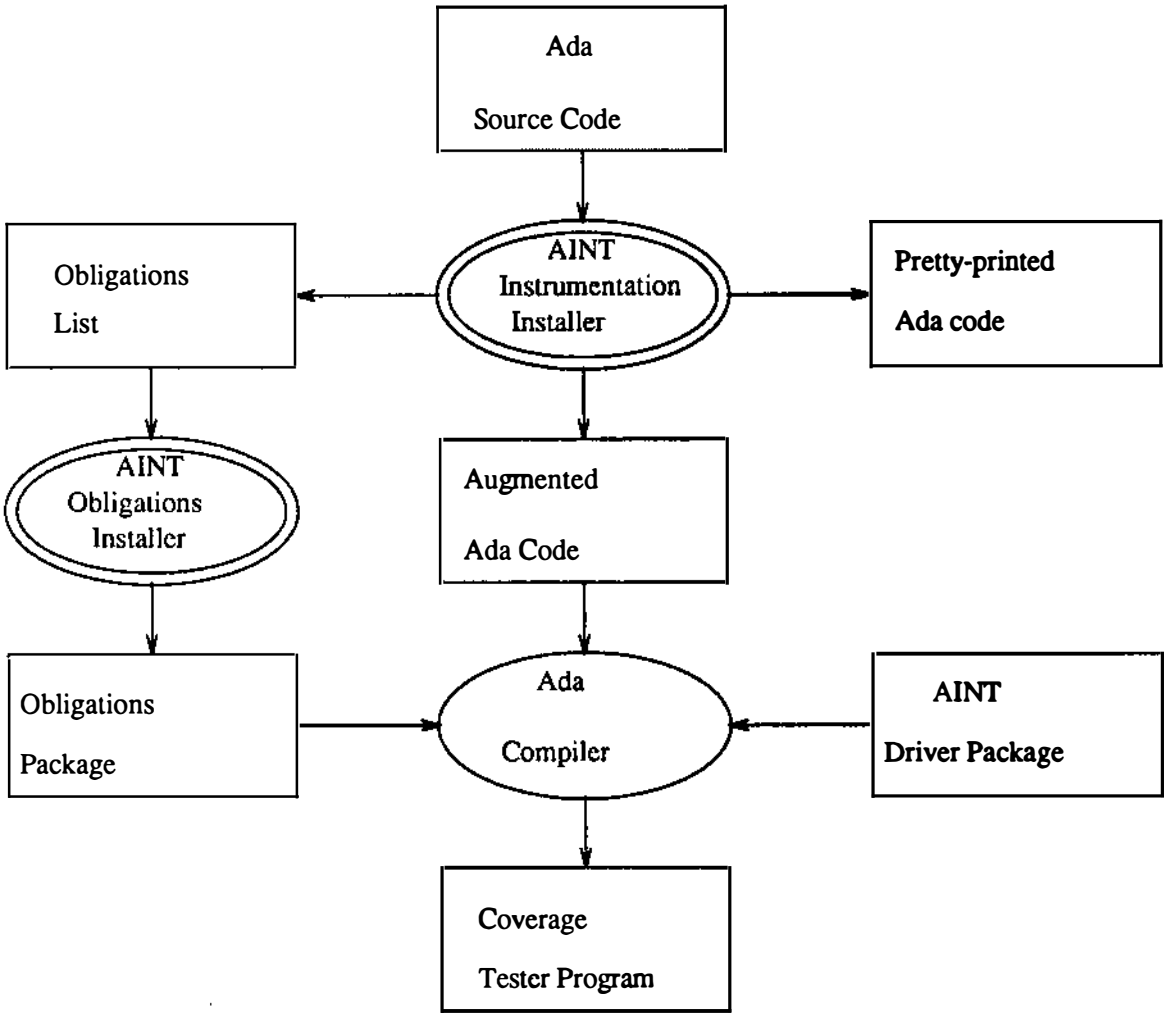


Figure 2: Architecture of AINT.

```
with Text_io; use Text_io;
procedure TOWERS_OF_HANOI is
    package IO_INTEGER is new INTEGER_IO ( INTEGER); use IO_INTEGER;
    NUM_OF_DISKS: NATURAL;
    procedure HANOI (N: NATURAL; X, Y, Z: CHARACTER) is
    begin
        if N /= 0 then
            HANOI(N-1, X, Z, Y);
            PUT("MOVE DISK");
            PUT(n); PUT("from");
            PUT(X); PUT(" to ");
            PUT(y); NEW_LINE;
            HANOI(N-1, Z, Y, X);
        end if;
    end HANOI;

begin
    PUT( "How many disks?"); NEW_LINE;
    GET( NUM_OF_DISKS);
    HANOI(NUM_OF_DISKS, 'X', 'Y', 'Z');
end TOWERS_OF_HANOI;
```

Figure 3: Towers of Hanoi.

worked by placing a call to the instrumenting package before every line of input code. These calls served to tell the package what statement was going to be executed next. For example, Figure 5 shows the augmented code for a recursive solution to the Towers of Hanoi program. The **Profile** and **ProfileIF** calls are to the instrumenting package **Probe** which keeps track of what statements and branches have been covered. The numbers in the calls indicate the number of the next statement to be executed and the strings indicate in what package the statement is contained.

The functionality of the input Ada code was not modified by AINT; however, there is an additional overhead of one procedure call per statement plus one call per conditional statement which results in increased execution time. Upon testing the software, this slowdown was determined to be excessive and a new approach was tried. Instead of placing procedure calls before every line of code, it was decided to put the instrumenting code there and avoid making any procedure calls not in the original code. In order to do this, the data structures of the instrumenting package had to be changed. Multidimensional arrays were used to keep track of obligation table information and to allow fast computation of what obligations had been covered. Figure 6 shows what the newer version of AINT will output for the Towers of Hanoi problem. **AINTstatement** and **AINTbranch** are boolean arrays in which the individual elements are set to true when the appropriate statement is encountered. The first array is two-dimensional with the first dimension indicating in which package the statement occurs and the second dimension indicating the statement number. The second array contains a third dimension which indicates which branch of the statement has been encountered. Initialization of these arrays is done in the **Probe** package which is created by the obligations installer. At the end of a test run, these arrays are examined to determine what obligations were not fulfilled.

Using this new approach, the instrumenting overhead is less substantial: one constant-time array

RUN 1:

```
How many disks?  2
MOVE DISK 1 from X to Y
MOVE DISK 2 from X to Z
MOVE DISK 1 from Y to Z

All statements covered.
All branches covered.
```

RUN 2:

```
How many disks?  0

Statement 11 not covered in "hanoi"
Statement 12 not covered in "hanoi"
Statement 13 not covered in "hanoi"
Statement 14 not covered in "hanoi"
Statement 15 not covered in "hanoi"
Statement 16 not covered in "hanoi"
Statement 17 not covered in "hanoi"
Statement 18 not covered in "hanoi"
Statement 19 not covered in "hanoi"
True branch of statement 10 not covered in "hanoi"
```

Figure 4: Sample output from execution of example program.

assignment per statement plus one constant-time array assignment per conditional. Although both versions added the same number of lines of code to the input program, the execution time of a single array assignment takes *much* less time than the execution of a procedure call along with the corresponding procedure code. Overhead for the first attempt at AINT increased with the number of statements executed; however, for most code, overhead for the second version is less than one hundred percent. Code which has many conditional statements and few or no procedure calls has slightly higher overhead.

While these figures sound poor at first, the second AINT implementation represents a design compromise. Remember that our objective is for developers to find *easy* ways to tailor instrumentation tools. The near one hundred percent increase in execution time is a fairly reasonable cost given that an execution event must be recorded for each program statement in order to obtain true statement coverage. It is reasonable to ask whether the statement execution obligations could be recorded on a *block*, not individual statement, basis, which is in fact how many other internal testing system are implemented. Unfortunately, when we considered how to perform block-level instrumentation, we discovered the grammar we were working with would have required our writing a much larger number of elaborate statements (to deal with such situations as handling of `GO TO` statements, and so on), all while being careful not to accidentally introduce unreachable code (e.g., by adding an obligations check after a `RETURN` statement.) The statement level instrumentation, on the other hand, was much simpler in design, and provided what was to us an acceptable testing performance.

Overall, the development of the AINT system took approximately 50 hours, from inception to demonstration of a distributable product to our funding agent. Most of this time was spent on the initial version of the translator, used to augment application programs. The remaining time was spent on extending the AINT system as described earlier, (informally) validating that our translator would perform reliably on a local test suite of Ada programs (used to ensure we provided suitable translations for code involving all Ada language constructs)[1], and finally checking the costs of addition instrumentation to a large application provided by our industrial funding agent, for use

[1] We regret to report that our grammar, which is independent of NewYacc, still fails to correctly parse Ada code involving static initialization of multi-dimensional arrays.

```
with Probe; use Probe;
with Text_io; use Text_io;
procedure TOWERS_OF_HANOI is
    package IO_INTEGER is new INTEGER_IO ( INTEGER);
    use IO_INTEGER;
    NUM_OF_DISKS: NATURAL;
    procedure HANOI (N: NATURAL; X, Y, Z: CHARACTER) is
    begin
        Profile(10,"hanoi");
        if N /= 0 then
            ProfileIF(1,10,"hanoi");
            Profile(11,"hanoi"); HANOI(N-1, X, Z, Y);
            Profile(12,"hanoi"); PUT("MOVE DISK");
            Profile(13,"hanoi"); PUT(N);
            Profile(14,"hanoi"); PUT("from");
            Profile(15,"hanoi"); PUT(X);
            Profile(16,"hanoi"); PUT(" to ");
            Profile(17,"hanoi"); PUT(Y);
            Profile(18,"hanoi"); NEW_LINE;
            Profile(19,"hanoi"); HANOI(N-1, Z, Y, X);
        ELSE
            ProfileIF(0,10,"hanoi");
        end if;
    end HANOI;

begin
    Profile(24,"hanoi"); PUT("How many disks?");
    Profile(25,"hanoi"); NEW_LINE;
    Profile(26,"hanoi"); GET(NUM_OF_DISKS);
    Profile(27,"hanoi"); HANOI(NUM_OF_DISKS, 'X', 'Y', 'Z');
end TOWERS_OF_HANOI;
```

Figure 5: Augmented version of Towers of Hanoi.

at their site. The DEC VMS -based implementation of our AINT system, running on a Microvax, added instrumentation code to an 18,000 source line Ada program in a matter of a few minutes.

4 IMPLEMENTATION OF AINT

In order to perform statement and branch coverage for input source code, an instrumenter needs to know what statements and branches must be executed and which are actually executed during a run. The way AINT accomplishes this is by building an obligations table and then marking statements and branches as they are executed in a test run. Upon completion of the run, any unmarked obligations are output.

Development of AINT was a three step process. The first step was to add the NewYacc actions to an Ada grammar so that the necessary statements and branches would be added to the obligations table and the input source code would be augmented with the appropriate instrumenting statements. For

```
with Probe; use Probe;
with Text_io; use Text_io;
procedure TOWERS_OF_HANOI is
    package IO_INTEGER is new INTEGER_IO ( INTEGER);
    use IO_INTEGER;
    NUM_OF_DISKS: NATURAL;
    procedure HANOI (N: NATURAL; X, Y, Z: CHARACTER) is
    begin
        AINTstatement(AINThanoi,10) := true;
        if N /= 0 then
            AINTbranch(AINThanoi,10,1) := true;
            AINTstatement(AINThanoi,11) := true; HANOI(N - 1, X, Z, Y);
            AINTstatement(AINThanoi,12) := true; PUT("MOVE DISK');
            AINTstatement(AINThanoi,13) := true; PUT(N);
            AINTstatement(AINThanoi,14) := true; PUT("from");
            AINTstatement(AINThanoi,15) := true; PUT(X);
            AINTstatement(AINThanoi,16) := true; PUT(" to ");
            AINTstatement(AINThanoi,17) := true; PUT(Y);
            AINTstatement(AINThanoi,18) := true; NEW_LINE;
            AINTstatement(AINThanoi,19) := true; HANOI(N - 1, Z, Y, X);
        ELSE
            AINTbranch(AINThanoi,10,0):= true;
        end if;
    end HANOI;

begin
    AINTstatement(AINThanoi,24) := true; PUT("How many disks?");
    AINTstatement(AINThanoi,25) := true; NEW_LINE;
    AINTstatement(AINThanoi,26) := true; GET(NUM_OF_DISKS);
    AINTstatement(AINThanoi,27) := true; HANOI(NUM_OF_DISKS, 'X', 'Y', 'Z');
end TOWERS_OF_HANOI;
```

Figure 6: Second augmented version of Towers of Hanoi.

each Ada statement type, three tasks were added as actions for the grammar rule:

1. a call is made to a procedure which enters the statement into the obligations table.
2. instrumenting code is added to the output.
3. the Ada statement is output by traversing the statement itself.

For example, the Ada grammar rule and NewYacc specification for an assignment statement is as follows:

```
sim_statement  ::=    assignment_stmt
                  [(AINT) probe() #1 bump()]
```

The `probe()` call adds the statement to the obligations table and outputs the appropriate instrumenting statement. The "#1" indicates a traversal of the assignment statement itself, and the

bump() increments a global statement counter.

Branch coverage required a little more creativity. To compute coverage for IF statements, two instrumenting statements were added per input IF statement, one for each possible branch. There are eight steps for the coverage of IF statements:

1. a call is made to a procedure which enters the true and false branch of the IF statement into the obligations table.
2. the **IF condition THEN** part of the statement is output.
3. the instrumenting code for the true branch is output.
4. the statements for the true branch are traversed and output (along with their associated statement instrumenting procedure calls).
5. the **ELSE** is output.
6. the instrumenting code for the false branch is output.
7. the statements for the false branch are traversed.
8. the **END IF** is output.

The following is a simplified version of how the above steps could be specified in NewYacc for an IF statement.

```
conditional    ::=   'IF' condition 'THEN' stmtlist 'ELSE' stmtlist 'END IF'
                     [(AINT) #1 #2 #3 probeif("then") #4 #5 probeif("else") #6 #7 #8]
```

The **probeif()** calls add the obligations to the table and output the appropriate code. The "#" symbols indicate traversal of the appropriate parts of the statement.

More complex augmentation must be done for **IF** statements without an associated **ELSE** branch, and for **IF** statements that use the **ELSIF** construction. In the first case, an **ELSE** branch which has only the single instrumenting statement is added in the augmented code. **ELSIF** requires more work, as the code is rewritten as cascaded **IF-THEN-ELSE** statements to simplify the computation of the branch coverage. Fig. 7(a) shows a small Ada program which contains both of these **IF** statement types. The augmented code (with statement instrumenting code omitted) for branch coverage is shown in Fig. 7(b).

The second development step involved creating a tool which would take the obligation lists as input and output the necessary obligation tables. To do this, a grammar written to parse the obligation lists was augmented with NewYacc statements in such a way that the entire package was generated by the tool. Figure 8 shows the data structures package which would be produced for the Towers of Hanoi program in Figure 6. This package initializes array elements for all of the statements and branches to be covered as false and all other statements and branches as true. The enumerated type **AINTtestnames** contains one element for each input package being covered and the constant **AINTdimensionlimit** gives the largest statement number in any of the input packages.

```
with Text_io; use Text_io;
procedure If_stmts(in X: NATURAL) is
begin
   IF x = 0 THEN
      x := x + 1;
   END IF;
   IF x < 3 THEN
      x := x + 3;
   ELSIF x < 5 THEN
         x := x + 5;
      ELSE
         x := x + 1;
   END IF;
end If_stmts;
```

(a)

```
with Text_io; use Text_io;
procedure If_stmts(in X: NATURAL) is
begin
   IF x = 0 THEN
      AINTbranch(AINTif_stmts,5,1) := true;
      x := x + 1;
   ELSE
      AINTbranch(AINTif_stmts,5,0) := true;
   END IF;
   IF x < 3 THEN
      AINTbranch(AINTif_stmts,8,1) := true;
      x := x + 3;
   ELSE
      AINTbranch(AINTif_stmts,8,0) := true;
      IF x < 5 THEN
         AINTbranch(AINTif_stmts,10,1) := true;
         x := x + 5;
      ELSE
         AINTbranch(AINTif_stmts,10,0) := true;
         x := x + 1;
      END IF;
   END IF;
end If_stmts;
```

(b)

Figure 7: Branch Coverage Augmentation.

In order to build this package, the obligations installer first outputs the header information and then traverses the input obligations file three times. The first traversal outputs the declarations for `AINTtestnames`, `AINTdimensionlimit` and the array type declarations. Then the obligation list is traversed twice; first to find all of the statements which must be executed in the packages and finally to find the branches in the packages. The `end probe;` is output at the end of the installer run.

As the final step in the development of AINT, it was necessary to write an Ada driver package that would call the Ada source code being tested, and then inform the tester of the results of the run. Computing the results of the run proved to be very simple, given the data structures used by the instrumenting process. Any element of the two instrumenting arrays which is false at completion time, signals an unfulfilled obligation.

5 CONCLUSION

The study of the software development process requires numerous tools to facilitate analysis. Automated tools are desirable for this study. Traditionally, this instrumentation and analysis has been performed *internally* by the language processors and execution environment. But this has made development of such tools time consuming, and, even worse, has made it difficult for ordinary users

```
package probe is
        AINTdimensionlimit : constant integer := 27;
        Type AINTtestnames is (AINThanoi);
        Type AINTstmtarray is array(AINTtestnames'First..
                                AINTtestnames'Last,1..AINTdimensionlimit) of boolean;
        Type AINTbrcharray is array(AINTtestnames'First..
                                AINTtestnames'Last,1..AINTdimensionlimit, 0..1) of boolean;
        AINTstatement: AINTstmtarray :=
                                AINTstmtarray'(AINThanoi => (10 => false, 11 => false, 12 =>
                                false, 13 => false, 14 => false, 15 => false, 16 => false,
                                17 => false, 18 => false, 19 => false, 24 => false, 25 =>
                                false, 26 => false, 27 => false, others => true));
        AINTbranch: AINTbrcharray :=
                                AINTbrcharray'(AINThanoi=> (10 =>(others => false), others =>
                                (others => true)));
end probe;
```

Figure 8: Obligations Package.

to change or adapt such tools for their own needs.

We have shown how *external* instrumentation and analysis can be performed with relative ease by ordinary users. Construction of the necessary language processors and execution environment is made easier by availability of translation tools such as NewYacc. The user may specify a desired source to source translation in terms of the constructs of the language being examined. These specifications are added to a grammar to create a parser in which the parse tree for input sentences can be traversed in ways specific to the application. Our experiences have shown that this adaptation can be performed with acceptable costs in terms of execution time.

Our claim concerning the utility of external adaptation has been supported by our presentation of AINT, an instrumentation and testing system for Ada code. AINT does simple branch and statement coverage analysis by performing a high-level source to source translation of the input code into a version of the code which, when used with the AINT system packages, does the computations necessary for the analysis. The development of this simple testing system demonstrates the ease with which other developers can build tools for different applications, including testing, debugging, and code profiling.

REFERENCES

[AIISu86]A. Aho, R. Sethi and J. Ullman. **Compilers: Principles, Techniques, and Tools.** Addison-Wesley Publishing Company. (1986).

[BaPII90]A. Babbitt, S. Powell and D. Hamlet. Prototype Testing Tools, Department fo Computer Science, Portland State University, TR-90-8, (June 1990).

[CaIt84] R. Cameron and M.R. Ito. Grammar-Based Definition of Metaprogramming Systems. **ACM Transactions on Programming Languages and Systems,** vol. 6, (January 1984), p. 20-54.

[Gilb77] T. Gilb. **Software Metrics,** Winthrop, (1977).

[Hals77] M. Halstead. **Elements of Software Science.** Elsevier North Holland, New York (1977).

[HoMi81] W. Howden and E. Miller, eds. **Tutorial: Software Testing and Validation Techniques,** IEEE, (1981).

[Huan78] J. Huang. Program Instrumentation and Software Testing, **Computer,** vol. 11, (April 1978), pp. 25-31.

[John79] S. Johnson. YACC: Yet Another Compiler Compiler. Bell Laboratories, (1979).

[Myer79] G. Myers. **The Art of Software Testing,** John Wiley & Sons, New York, (1979).

[PuCa89] J. Purtilo and J. Callahan. Parse Tree Annotations. **Communications of the ACM,** vol. 32, (December 1989), pp. 1467-1477.

[Watt87] D. Watt et al. **Ada Language and Methodology.**, Prentice-Hall, (1987).

Research was begun while supported by Office of Naval Research contract, and is now supported by DARPA/ISTO under its Common Prototyping Language initiative.

Applying the Category–Partition Method to C++

Keith Koplitz

Mentor Graphics Corporation
8500 SW Creekside Place
Beaverton, Oregon 97005
(503) 626–7000

Keywords

software testing, functional testing, test specification, automatic test generation, category–partition method

Abstract

We have applied the category–partition method for specifying functional tests within a C++ development environment. It replaced our prior use of ad hoc methods with a more disciplined approach to testing. The tool–based implementation provided us with an automated, repeatable test process. The formal test specifications facilitated the reuse of previous work. This was essential to our managing the changes associated with an evolutionary software development model. Since applying this test methodology, we have found the test coverage levels to be consistently higher.

Biographical Sketch

Keith Koplitz is a Software QA Engineer in Mentor Graphics' Advanced Products Division. He is currently testing an object–oriented database for electronic system design. His primary interest is software engineering with a focus on testing and metrics. He has B.S. degrees in Architectural Studies and Computer Science from Washington State University.

Introduction

We have applied the category–partition method [1] for specifying functional tests within a C++ development environment. It replaced our prior use of ad hoc methods with a more disciplined approach to testing. The tool–based implementation provided us with an automated, repeatable test process. The formal test specifications facilitated the reuse of previous work. This was essential to our managing the changes associated with an evolutionary software development model [2]. Since applying this test methodology, we have found the test coverage levels to be consistently higher.

The first section of the paper discusses our need for a new testing methodology. The second section describes how we applied the category–partition method. The section starts by outlining the general steps of the method. Then, a detailed example of using the method is presented. The third section is a summary of key results. The final section provides a wrapup of the main points.

Background

We started test planning for the Design Data Management System (DDMS), an object–oriented design database, in June, 1988. The planning highlighted our need to satisfy two primary customer demands: to increase functionality and to reduce the time to market. This meant we had more to test and less time to test it.

In addressing these demands, two key decisions affected our approach to functional testing: the use of C++ and the use of an evolutionary software development model.

Using C++ was relatively new to our project team. Previous experience was limited to implementing base utilities and data structures. We had even less experience with how to effectively test an object–oriented system.

Using the evolutionary software development model presented an even greater concern. Since all DDMS requirements could not be determined at the start, we had to make phased releases to internal product groups (clients). Each release enhanced our client's understanding of the system and allowed them to refine their requirements. Also, the releases allowed our clients to start their software development earlier.

Our testing concerns centered on the conflicting criteria for these phased releases and the final product. For the phased releases, we had to determine if our clients could do "useful" work with the software. This meant anticipating the most likely conditions (breadth) under which it would be used. The final product had to satisfy our customer's requirements for reliability. This required consideration of all the conditions (depth) affecting the behavior of the software. With each release, test specifications and test cases had to be revised.

Existing ad hoc methods did not fit this development model. Previously, most test specifications were written without using any particular format and did not support traceability. The only automated task was test case execution. We needed a rigorous methodology for the basis of a repeatable test process. The category-partition method (CPM) satisfied this need.

Applying CPM to C++

Partitioning System Functionality

One starts the category-partition method by partitioning system functionality. This decomposition continues until manageable units have been identified. We found that the object-oriented architecture of DDMS, and its implementation in C++, provided a natural breakdown. The objects were manageable units and the methods on these objects represented the functionality to be tested. As shown in Figure 1, the resulting class derivation hierarchy could be traversed to examine these functional units.

Given a function, the next step is identifying the parameter and environmental states (or conditions) that affect its behavior. For each condition, categories are defined that characterize the possible values. Using equivalence partitioning [3], we would divide a category's domain into classes of values that are expected to produce the same result in the function. From these classes, the actual values for the test cases were selected.

Finally, the interaction between category choices is defined. Properties are assigned to specific choices and then selector expressions are used to constrain possible value combinations. The resulting formal specification fully characterizes the behavior of the function.

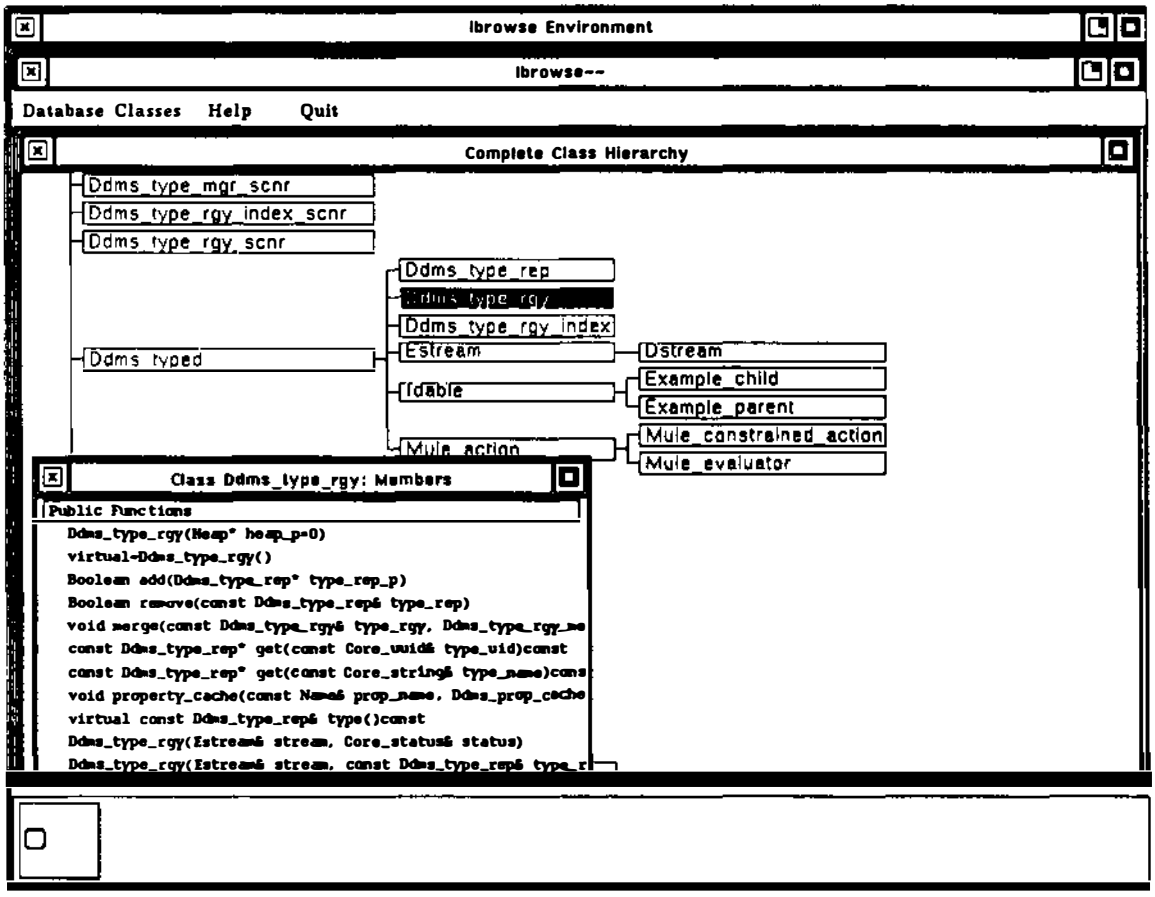


Figure 1. Functional Partitioning Using a C++ Code Browser.

Revising the Test Process

Early planning had revealed the need to improve the existing test process. CPM offered a systematic approach to specifying functional tests. Now, we needed to merge this methodology with our C++ development environment. The goal was to create an automated, repeatable test process.

After a brief survey, we found that most of the new process could be implemented using existing tools. This approach benefitted us in two ways. First, the existing tools were supported. We would not have to find the resources to maintain a new set of tools. Second, the time to realize the process would be minimized. Together, these benefits reduced the risk of using the new approach.

The revised process and its data flow is depicted in Figure 2. The supporting tools had to provide three main capabilities; extraction, representation, and generation. First, the class definitions had to be extracted from the source files. Next, a consistent format was required to represent the test specifications. Finally, a means for generating test conditions was needed.

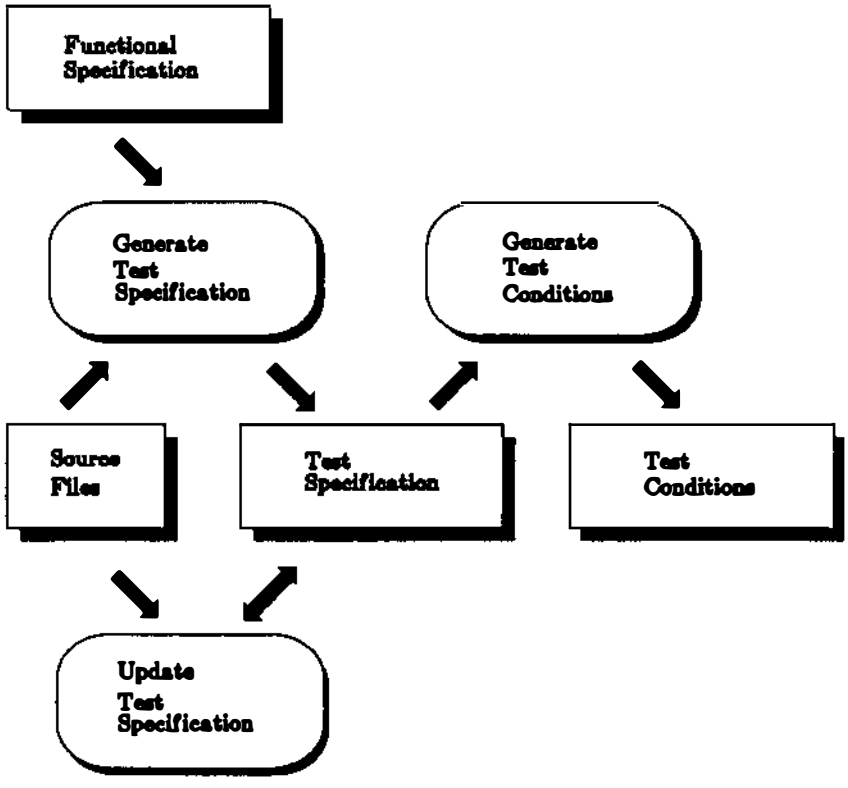


Figure 2. Revised Test Process Using Category–Partition Method.

We had both a code browser and a quick reference tool for extracting class definitions from C++ header files. These files formally specify each classes' protocol or external interface. Automating this step was essential, since these files were expected to change as we made phased releases.

For another project, a grammar-based generator had been developed for language testing. We found the rules syntax flexible enough to specify the behavior of functions. The strings generated from the grammar represented the test conditions for the given function. The constraining of test conditions was the one part of CPM that we could not support with an existing tool. We decided to use the UNIX™ software tools LEX and YACC to implement a parser for the needed constraint language.

The use of CPM and the related tools are described in the following example.

Generating Test Specifications

The first step of the new testing process was to generate a functional test specification. After selecting a class to test, we extracted the pertinent information about the class.

Given a C++ header file, the existing quick reference tool would list the name of the class and its public functions (external interface). An example of the tool's output is shown in Figure 3.

```
CLASS: Ddms_type_rgy

PUBLIC DATA: None

PUBLIC FUNCTIONS:
Ddms_type_rgy(Heap* heap_p)
Ddms_type_rgy(Stream& stream, Status& status)
virtual ~Ddms_type_rgy()
Boolean add(Ddms_type* type_p)
Boolean remove(const Ddms_type& type)
void merge(const Ddms_type_rgy& type_rgy, Type_rgy_merge_type merge_type)
Ddms_type* get(const Identity& type_identity) const
Ddms_type* get(const String& type_name) const
void property_cache(const Property_name& prop_name, Propery_cache_type cache_type)
virtual void write_typed(Stream& stream, Status& status)
```

Figure 3. Quick Reference for Type Registry Class.

The tool listed this information for each of the classes defined within the file. A script was written to transform the quick reference text into a test specification template. Using AWK, the output was reformatted into the grammar rules syntax. Figure 4 shows the specification block for a class function. A similar block would be created for each of the classes' functions.

```
class Ddms_type_rgy

function Boolean add(Ddms_type* type_p)
start        ::= environment add parameters result
environment  ::= { ENVIRONMENT-LIST }
parameters   ::= { PARAMETER-LIST }
result       ::= return= VALUE
```

Figure 4. Test Specification Template for Type Registry Class.

Automating this extraction step allowed us to consistently apply our testing strategy. We had decided that the most practical approach was to directly test only the public functions. We relied upon test coverage analysis to indicate what we were covering in the private implementation. This allowed us to use our existing test driver technology without requiring any special hooks. In essence, we were writing sample applications or design tools.

By using the code browser and quick reference tool, we could effectively partition DDMS into its functional units. This satisfied the first step of the category-partition method. The next step involved consolidating information about each function and its specified behavior.

Initially, we would start with functional, or "black-box", techniques to write the test specification. Then, as the process was repeated for a given class, we would rely more upon structural, or "white-box", techniques to update the function's test specification.

```
Environment Conditions:
    Type Registry State
        Specified type already exists in registry
        Registry is not caching types by properties
        Registry is caching types by property names
        Registry is caching types by property names and values

    Type State
        Type has a property list
        Type does not have a property list

Parameters:
    Type Pointer
        Null Pointer
        Pointer to a current version of the type
        Pointer to a newer version of the type
        Pointer to an older version of the type

Results:
    Return Values
        TRUE
        FALSE
```

Figure 5. Environment, Parameter, and Result Categories for Add Function.

We used the functional specification, design documentation, and class definitions to identify the categories for the function's parameters and environment conditions. Figure 5 shows a breakdown of categories and value choices is shown for a function. The grammar representation of the test specification allowed us to also specify expected results for the function. This was not supported in the original presentation of CPM. Later work described in [4] removed this restriction.

Figure 6 shows the categories and their choices written using the rules syntax for the grammar. At this time, enough of the test specification has been completed to allow test conditions to be generated. However, this "unconstrained" specification does not consider the interactions between the different category values. Thus, the generated value combinations usually contain contradictions.

```
class Ddms_type_rgy

function Boolean add(Ddms_type_rgy* type_p)
start              ::= environment add parameters result

environment        ::= { type_rgy_state , type_state }
type_rgy_state     ::= type_exists
type_rgy_state     ::= rgy_not_cached
type_rgy_state     ::= rgy_cached_by_name
type_rgy_state     ::= rgy_cached_by_value

type_state         ::= prop_list
type_state         ::= no_prop_list

parameters         ::= ( type_p )
type_p             ::= null
type_p             ::= current_version
type_p             ::= newer_version
type_p             ::= older_version

result             ::= return= TRUE
result             ::= return= FALSE
```

Figure 6. Unconstrained Test Specification for Add Function.

Generating Test Conditions

An "unconstrained" test specification is often incomplete. It only describes function behavior relative to the possible values for its parameter, environment, and result conditions. It does not consider the interaction between these values. As a result, the generated test conditions usually contain contradictions. For example, Figure 7 shows conditions that differ only in their expected result value. This is clearly a contradiction.

```
{ rgy_not_cached , prop_list } add ( current_version ) return=FALSE
{ rgy_not_cached , prop_list } add ( current_version ) return=TRUE
{ rgy_cached_by_name , no_prop_list } add ( current_version ) return=FALSE
{ rgy_cached_by_name , no_prop_list } add ( current_version ) return=TRUE
```

Figure 7. Test Conditions Generated from an Unconstrained Test Specification.

The constraint language is primarily used to specify these interactions and thus remove contradictions from the generated test conditions. Another use of contraints is filtering redundant value combinations. It is often sufficient to use a given category value once, especially for error conditions. By using the constraints, we can restrict the use of given values and document our test design decisions.

```
operation add ( type:Ddms_type in )
{
    if type exists and is same version then
        return FALSE
    end if

    add type to identity registry
    if type is not transitional then
        add type to name registry
    end if

    if registry is not caching types then
        return TRUE
    end if

    ...

    return TRUE
}
end add
```

Figure 8. Structured Design Description for Add Function.

We start the constraint step by reviewing the function's specified behavior. Initially, we use the functional specification or a structured design description, like Figure 8, to understand its behavior. Later, we often use the implementation to help clarify the interactions between category values. Typically, we record information about these interactions at the same time that we identify the appropriate category values.

```
environment     ::= { type_rgy_state , type_state }
type_rgy_state  ::= type_exists                 [property exists] (A)
type_rgy_state  ::= rgy_not_cached
type_rgy_state  ::= rgy_cached_by_name
type_rgy_state  ::= rgy_cached_by_value

parameters      ::= ( type_p )
type_p          ::= null                        [property null] (D)
type_p          ::= current_version
type_p          ::= newer_version               [property transitional] (B)
type_p          ::= older_version               [property transitional] (B)

result          ::= return= TRUE
result          ::= return= FALSE               [if exists][if not transitional] (C)
result          ::= return= FALSE               [if null] (E)
```

Figure 9. Constraining FALSE Result Value for Add Function.

In Figure 9, the grammar rules have been annotated to constrain the use of the FALSE value. From reviewing the design description, we knew that the function should return FALSE when the specified type already exists in the registry (see *(A)*) and is not a different version *(B)*. Therefore, we assigned a property to each of the related rules. A selector expression *(C)* using these properties was then added to qualify the selection of the FALSE value.

While trying to complete this constraint, we found that it was not specified how the function would handle a null type pointer. This highlights another benefit of using this method. The more disciplined approach helps us identify where the functional specification is incomplete or ambiguous.

After reviewing the updated functional specification, we added another property *(D)* and selector expression *(E)* to the test specification. This same technique is applied until all the value interactions have been specified. The final, "constrained" version of the test specification is shown in Figure 10. Note that the selector expressions for the FALSE value have been merged into a single rule.

```
class Ddms_type_rgy

function Boolean add(Ddms_type_rgy* type_p)
start              ::= environment add parameters result

environment        ::= { type_rgy_state , type_state }
type_rgy_state     ::= type_exists          [property exists]
type_rgy_state     ::= rgy_not_cached       [property not_exists,not_cached]
type_rgy_state     ::= rgy_cached_by_name   [property not_exists,name_cached]
type_rgy_state     ::= rgy_cached_by_value  [property not_exists]

type_state         ::= prop_list
type_state         ::= no_prop_list         [if name_cached]

parameters         ::= ( type_p )
type_p             ::= null                 [if not_exists and not_cached][property null]
type_p             ::= current_version      [property not_null]
type_p             ::= newer_version        [if exists][property not_null,transitional]
type_p             ::= older_version        [if exists][property not_null,transitional]

result             ::= return= TRUE         [if not_null][if not_exists or transitional]
result             ::= return= FALSE        [if null or exists][if not transitional]
```

Figure 10. Constrained Test Specification for Add Function.

Figure 11 shows the test conditions generated from the constrained test specification. Each string is a concise description of the values and states that will be used to test the function. Test cases are defined by selecting one or more of the conditions from a related set of functions.

```
{ rgy_not_cached, prop_list } add ( current_version ) return= TRUE
{ rgy_not_cached, prop_list } add ( null ) return= FALSE
{ rgy_cached_by_name, prop_list } add ( current_version ) return= TRUE
( rgy_cached_by_name, no_prop_list } add ( current_version ) return= TRUE
{ type_exists, prop_list } add ( current_version ) return= FALSE
{ type_exists, prop_list } add ( older_version ) return= TRUE
{ type_exists, prop_list } add ( newer_version ) return= TRUE
{ rgy_cached_by_value, prop_list } add ( current_version ) return= TRUE
```

Figure 11. Generated Test Conditions for Add Function.

The test cases are implemented using command scripts. These scripts are run using test drivers (or exercisers) that map the commands into calls of the member functions. A typical test case implementation is shown in Figure 12.

After executing the tests, we analyze statement-level test coverage data to determine the effectiveness of the testing. For the add function, we found that using ad hoc methods resulted in only 69% of the function's statements being executed. Applying CPM resulted in test cases that covered 100% of the statements.

```
#
# TEST CASE NAME xddms_type_rgy_3
#
# SPECIFIC DESCRIPTION
#
# PR/PER NUMBER AND SYNOPSIS
#
# INPUTS / OUTPUTS
# { rgy_not_cached, prop_list } add ( current_version ) return= TRUE
# { exists  } get ( type_name ) return= type_rep_p
# { exists  } get ( type_uid ) return= type_rep_p
#
# COMMENTS
#
type_manager
new_identity
new_type_rep 0 DFD dynamic 0 false
new_type_rgy
add_type_rgy 0 9
write_type_rgy 0 /tmp/xddms_type_rgy/types3.rgy
delete_type_rgy 0 false
read_type_rgy /tmp/xddms_type_rgy/types3.rgy
get_by_name_type_rgy 0 DFD
get_by_identity_type_rgy 0 0
same_type_rep 10 11
delete_type_rgy 0 true
delete_type_rep 9
delete_type_rep 10
delete_type_rep 11
delete_identity 0
exit
```

Figure 12. Implemented Test Case for Add Function.

Updating Test Specifications

With the ad hoc methods, updating test documentation and test suites was always a tedious, time-consuming task. We often found that the original test selection and design rationale had not been retained. When the information was available, it was in a form that usually required extensive rework. Since the evolutionary development model would mean even more changes, we needed to ensure that we could effectively manage software changes.

By using CPM, we were able to automate some of the update task. The formal test specifications helped us to retain most of the test rationale. Also, the grammar format allowed us to make specific changes for a given function and then, quickly regenerate the desired test conditions. With the addition of a simple text filter, we could use the quick reference tool to assess the impact of software changes. First, we would extract the function definitions (or prototypes) from both the source files and the test specifications. Then, we compared the results to determine what changes had been made to the classes' external interface.

By using the test condition strings to document the test procedures, we could check the status of current test implementation or even, determine what test cases needed to be updated.

Summary

- CPM and the described test process has been used on five projects. It was the primary methodology on two projects and the secondary methodology on the remaining projects.
- DDMS is about 30K Non-Commentary Source Lines with 90 classes.
- 6000 lines of test specification have been written for DDMS.

Conclusion

We have successfully used the category-partition method and a supporting toolset to verify complex object-oriented software systems. The testing methodology was adopted to reduce the risk of working in a new language and developing software in a highly dynamic environment. We measured our success in terms of satisfying the original objectives for such a methodology.

First, we were able to develop an automated, repeatable test process. By automating most of the tedious tasks, we were able to focus more effort on formally specifying and understanding the behavior of the functions being tested. The integration of the tools made it easy to repeat process steps when necessary.

Second, the better documentation of test specification and design rationale improved the maintainability of the test suites. When software changes were noted, we could go back and clearly identify how the existing tests were affected by the changes. For one particular project, a new engineer was able to resume the efforts of a former engineer in just a couple weeks.

Finally, initial statement-level test coverage has been consistently higher for the new methodology. Comparing data from first time test runs has shown that CPM generated tests provide 70-80% coverage, while ad hoc generated tests only provide 50-60% coverage. Applying CPM positions us to meet or exceed our test coverage targets with the initial test run.

References

1. Ostrand, T.J. and Balcer, M.J. The category-partition method for specifying and generating functional tests. *Communications of the ACM* 31,6 (June 1988),676-686.

2. Combest, R. and Strater, S. FI^{3}T: An evolutionary approach to software development at Mentor Graphics. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference.* (September 1988, Portland, Oregon). 143-155.

3. Myers, G.J. *The Art of Software Testing.* John Wiley & Sons, Inc. New York, New York. 1979.

4. Balcer, M.J. *et al.* Automatic generation of test scripts from formal test specifications. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification.* (December 1989, Key West, Florida). 210-218

Tools for Testing Object-Oriented Programs *

Phyllis G. Frankl

Roong-Ko Doong

Department of Computer Science
Polytechnic University
333 Jay Street
Brooklyn, NY 11201
(718-260-3870)
e-mail: pfrankl@polyof.poly.edu

Keywords: software testing, object-oriented, abstract data type

ABSTRACT

This paper describes a new approach to testing object-oriented programs, and a set of tools based on this approach. Test cases consist of pairs of sequences of methods, along with a tag indicating whether those sequences should put objects of the class under test into the same abstract state. Experimental prototypes of tools for test generation and test execution are described. The test generation tool requires the availability of an algebraic specification of the abstract data type being tested, but the test execution tool can be used when no formal specification is available. We also discuss how inheritance affects the testing process.

Biographical information

Phyllis G. Frankl received the B.A. degree in Mathematics and Physics from Brandeis University in 1979, the M.A. degree in Mathematics from Columbia University in 1981, and the M.S. and Ph.D. degree in Computer Science from New York University in 1985 and 1987, respectively. She has been an Assistant Professor of Computer Science at Polytechnic University since 1987. Her current research involves various aspects of software testing, including evaluation of the effectiveness of data flow testing, testing object-oriented programs, and related issues in software specification and semantic analysis of programs.

Roong-Ko Doong received the B.S. degree in Electrical Engineering from National Cheng Kung University, Taiwan in 1984, the M.S. degree in Computer Science from Florida Institute of Technology, 1987 . He is currently working toward a Ph.D. degree in Computer Science at Polytechnic University.

*This research was supported in part by NSF grant CCR-8810287 and by the New York State Science and Technology Foundation.

1 Introduction

In recent years, object-oriented programming has emerged as a new software development paradigm which has many attractive software engineering features. While a considerable amount of research has focussed on how to design, implement and re-use object-oriented software, relatively little work has addressed the question of, "how should object-oriented software be tested?" This paper presents a new approach to testing object-oriented programs and describes experimental prototypes of two tools based on this approach, a test generation tool and a test execution tool.

Most research on software testing has implicitly assumed that the units under test are control abstractions, such as procedures and functions. In object-oriented programs, the emphasis is shifted from control abstractions to data abstractions. The program units one wishes to test are *classes* which implement abstract data types, rather than individual procedures or functions. Rather than focussing on input and output, we would like to test whether particular sequences of messages put an object into the right state.

In section 2 of this paper, we describe a new model of the testing process which makes this shift of emphasis. This model leads in a natural way to development of tools for test generation and test execution. In section 3 we describe experimental prototypes of such tools, and illustrate their use with examples. Our model is inspired by the theory of algebraic specifications of abstract data types, and our test generation tool uses an algebraic specification as input; however the test execution tool does not require the availability of a formal specification. Section 4 discusses some of the issues raised by inheritance, and section 5 compares our approach to related work, and points out several directions for future work. In the remainder of this introduction, we review relevant background material on software testing, object-oriented programming languages, and algebraic specifications of abstract data types.

1.1 Background on Software Testing

It is well known that testing is one of the most time-consuming parts of the software development process. Increased automation of the testing process could lead to significant saving of time, thus allowing for more thorough testing. Three aspects of the testing process which could potentially be at least partially automated are test data generation, test execution, test checking. Our approach to testing object-oriented programs involves all three of these areas.

One area for potential automation is in the construction of test drivers. Many testing methods can be applied to individual subprograms. When the program unit being tested is a whole program, the inputs and outputs are usually sets of files. When the unit being tested is a procedure or function the inputs and outputs may include values of parameters and of global variables, as well as values read from and written to files. In order to test a procedure, it is necessary to build a *driver* program which initializes global variables and actual parameters to the appropriate values, then calls the procedure, then outputs final values of relevant globals and parameters. It can be quite cumbersome to initialize the inputs and check the values of the outputs. It is particularly unwieldy if, as is often the case in object-oriented programming, the parameters have complicated types. The model described below for testing object-oriented programs circumvents this problem.

Another problem which arises in testing software is the *oracle problem* – after running P on a test case, it is necessary to check whether the result agrees with the specification. This is often a non-trivial problem, for example if there is a great deal of output, or if it is difficult to calculate the correct value [WEY82]. Our testing method uses a novel approach which allows the correctness of test cases to be checked automatically by the test execution system.

1.2 Overview of Object-Oriented Programming

Object-oriented languages support *abstract data types* and *inheritance*. An abstract data type is an entity which encapsulates data and the operations for manipulating that data. In object-oriented programming,

the programmer writes *class* definitions, which are implementations of abstract data types. An *object* is an instance of a class; it can be created dynamically by the instantiation operation, usually called "new" or "create". A language supports *inheritance* if classes are organized into a directed acyclic graph in which definitions are shared, reflecting common behavior of objects of related classes.

A class consists of an *interface* which lists the operations which can be performed on objects of that class and a *body* which implements those operations. The state of an object is stored in *instance variables* (sometimes called *attributes*), which are static variables, local to the object. An object's operations are sometimes called *methods*.

In object-oriented programs, computation is performed by sending *messages* to objects. A message invokes one of the object's methods, perhaps with some arguments. The invoked method may then modify the state of its object and/or send messages to other objects. When a method completes execution, it returns control (and in some cases returns a result) to the sender of the message.

The inheritance mechanism of object-oriented languages facilitates the development of new classes which share some aspects of the behavior of old ones. A descendent (subclass) C_d of a class C inherits the instance variables and methods of C. C_d may extend the behavior of C by adding additional instance variables and methods, or specialize C by redefining some of C's methods to provide alternative implementations (or some combination thereof).

A dynamic binding mechanism is used to associate methods with objects. It is legal to assign an object of class C_d to a variable of class C (but not vice-versa). After doing so, a message sent to this object will invoke C_d's method. For example, consider a class polygon with subclasses triangle and square, each of which redefines polygon's perimeter method. Assigning an object of class square to a variable of class polygon, then sending the perimeter message will invoke square's perimeter method. This provides for polymorphism.

Some examples of object-oriented languages include Smalltalk, C++, and Eiffel [GOL83, STR86, MEY88]. While Ada and Modula-2 are not, strictly speaking, object-oriented languages, they do provide support for data abstraction; thus, some of the ideas discussed here are relevant to them. See [MEY88] for a good overview of the object-oriented approach and cogent arguments that object-oriented programs are easier to design, implement, maintain, and re-use than traditional programs.

1.3 Algebraic Specifications of Abstract Data Types

Before we can talk about testing a class, we must have some concept of what it means for the class to be correct. Thus, we must have some means, formal or informal, of specifying the abstract data type which the class is intended to implement. We now give an overview of a formal specification technique known as algebraic specification, upon which some of our testing techniques are based.

Algebraic specification techniques [GOG78, GUT77, LIS75] describe a data abstraction by describing the interaction between the operations, without reference to the underlying representation. An algebraic specification of an ADT has a syntactic part and a semantic part. The syntactic part is a list of function names, and their signatures (the types which they take as input and produce as output). The syntactic specification is similar to the interface specification part of a class definition in an object oriented language.

The semantic part of the specification consists of a list of axioms describing the relation between the functions. Some specification techniques allow for a list of preconditions describing the domains of the functions, while others allow functions to return error values indicating that a function has been applied to an element outside of its domain.

In the specification of a type T, a function `f` is called an *observer* if it's output is a type other than T, and either a *constructor* or *transformer* if T is an output of `f`. Constructors and transformers change the state of the ADT, while observers allow the "outside world" to query the ADT about its current state. The distinction between constructors and transformers depends on the semantic part of the specification: `f` is a transformer if it is possible to eliminate `f` from any sequence by applying rewrite rules, and is a constructor otherwise.

```
CLASS SET_OF_INT EXPORT
  Create, Insert, Delete, Empty, Member
CONSTRUCTOR
  Create;
    -- -- Create an empty set
  Insert(x:  INTEGER)
    -- -- Insert x into set
TRANSFORMER
  Delete(x:  INTEGER)
    -- -- Delete x from set
OBSERVER
  Member (x:  INTEGER): BOOLEAN;
    -- -- Is x in the set?
  Empty:  BOOLEAN
    -- -- Is set empty?
VAR
  old:  SET_OF_INT;
  a, b:  INTEGER
AXIOM
    -- -- 1
  Create.Empty → TRUE;
    -- -- 2
  old.Insert(a).Empty → FALSE;
    -- -- 3
  Create.Member(a) → FALSE;
    -- -- 4
  old.Insert(a).Member(b) → if a = b then TRUE
            else old.Member(b)
    -- -- 5
  Create.Delete → Create;
    -- -- 6
  old.Insert(a).Delete(b) → if a = b then old.Delete(b)
            else old.Delete(b).Insert(a)
END
```

Figure 1: Specification of SET_OF_INT

For example, the specification in Figure 1 describes a set of integers ADT. The specification is written in LOBAS (Language for Object Based Algebraic Specification) [DOO90], a language which we are developing in tandem with our study of testing object-oriented programs. The syntax of LOBAS is similar to that of object-oriented programming languages: the type being specified is an implicit input to each operation except `Create`; sequences of operations read from left to right with dots as separators. This specification says that type SET_OF_INT has a constant `Create` of type SET_OF_INT (the empty set), and operations

`Insert`: SET_OF_INT × Integer ⟶ SET_OF_INT

`Delete`: SET_OF_INT ⟶ SET_OF_INT

`Member`: SET_OF_INT × Integer ⟶ Boolean

`Empty`: SET_OF_INT ⟶ Boolean.

Axioms 1,2,3, and 5 say that `Create` is empty, the set obtained by inserting any item in any set is not empty, no element is a member of `Create`, and the result of deleting any element from `Create` is `Create`, respectively. Axiom 4 says that if an element `a` is inserted into a set `old`, then `b` is a member of the resulting set if and only if either `b` = `a` or `b` is a member of `old`. Axiom 6 says that the result of inserting `a` then deleting `b` from a set `old` is the same as the result of deleting `b` from `old`, if `a` = `b`; if `a` ≠ `b` it is the same as deleting `b` from `old` then inserting `a`. Note that axiom 6 says that if duplicate copies of element `b` have been

inserted into `old`, they are all deleted. Thus, the fact that duplicates are present can never affect the result of a sequence of operations ending in an observer, so this ADT behaves like a set, rather than a bag.

Our notion of test cases for data abstractions is inspired by the formal semantics of algebraic specifications developed by [GOG78, GUT77], which involves modelling a specification by a heterogeneous word algebra. Roughly speaking, two sequences `S1` and `S2` of operations returning a value of type T are equivalent if, for any observer `O`, we can use the axioms as rewrite rules, to transform `S1.O` to `S2.O`. For example, `Create.Insert(5).Insert(3).Delete(5)` is equivalent to `Create.Insert(3)` because we can apply axiom 6 twice to give

`Create.Insert(5).Insert(3).Delete(5)` ⟶
`Create.Insert(5).Delete(5).Insert(3)` ⟶
`Create.Insert(3)`.

It follows that for any observer `O` we can rewrite

`Create.Insert(5).Insert(3).Delete(5).O` to
`Create.Insert(3).O`.

Intuitively, each equivalence class of sequences returning type T corresponds to a different state of the ADT. For example, the two above sequences represent the state in which the set has a single element, 3.

A specification of type T is sufficiently complete if any sequence of operations ending in an observer of type T' can be simplified into a constant of type T'. A specification is inconsistent if there is a sequence of operations ending in an observer which can be simplified into two different elements, by applying rewrites in different orders; it is consistent otherwise. Throughout this paper, we will only consider specifications which are consistent and sufficiently complete.

We next consider what it means for a class C to be a correct implementation of an ADT T. The idea is that C is correct if and only if we can map operations of T to operations in C in a way which preserves signatures and preserves the way operations transform states. We will say that objects O_1 and O_2 of class C are *observationally equivalent* if and only if for any sequence S of operations of C ending in an observer, $O_1.S$ is equivalent to $O_2.S$ (in the output type of S). Thus, O_1 is observationally equivalent to O_2 if and only if it is impossible to distinguish O_1 from O_2 using the operations of C. Two observationally equivalent objects are in the same "abstract state", even though the details of their representations may be different.

DEFINITION:
A class C is a correct implementation of an abstract data type T if and only if for every pair (S_1, S_2) of sequences of operations in T,

> S_1 is equivalent to S_2 if and only if the corresponding sequences of operations in C put objects of class C into observationally equivalent states.

2 An Object-Oriented Notion of Test Case

Traditional testing strategies are based on viewing a specification and a program as functions mapping an input space to an output space. Let $\mathcal{D}$ be the domain of a specification $\mathcal{S}$. Program $\mathcal{P}$ is a *correct* implementation of $\mathcal{S}$ if and only if $\mathcal{P}$ and $\mathcal{S}$ compute the same function on $\mathcal{D}$, i.e. iff $(\forall d \in \mathcal{D})(\mathcal{P}(d) = \mathcal{S}(d))$. In order to test $\mathcal{P}$, one chooses a subset $\mathcal{T} \subseteq \mathcal{D}$, runs program $\mathcal{P}$ on each element of $\mathcal{T}$, and compares the output to the specified value. If $\mathcal{P}(t) \neq \mathcal{S}(t)$ for some $t \in \mathcal{T}$, then an error has been found. On the other hand, if the program agrees with the specification on each element of a well-chosen test set $\mathcal{T}$, our confidence that the program is correct should increase. Of course nothing less than exhaustive testing ($\mathcal{T} = \mathcal{D}$) can *guarantee* correctness.

In one of the early papers on specification of data abstractions [LIS75], Liskov and Zilles pointed out that it is possible to specify a data abstraction by specifying the intended input-output behavior for each of its operations individually, but doing so is usually cumbersome and leads to overspecification of the underlying representation of the data. Similarly, it is possible to test a class by testing each of it's methods individually, but doing so may necessitate the construction of complicated drivers and output checking mechanisms. For example, a test case for the push operation in a stack would consist of stack and an item and the output would be another stack. Thus the driver would have to initialize the input stack and checking the output would entail examining the output stack to see if it is the correct result. Moreover, testing individual operations separately shifts the focus of testing away from the essence of the data abstraction: the interaction between operations resulting in the equivalence of various sequences of messages.

Our approach to testing classes is based on a different kind of test case, which, we believe, is more appropriate for testing data abstractions. It is inspired by the definition of *correctness* given above. A test case for C will consist of a pair of sequences of operations (messages), along with a tag indicating whether or not they should be equivalent, according to the specification. For example,

`(Create.Insert(5).Insert(3).Delete(5), Create.Insert(3), equivalent)`, and

`(Create.Insert(5).Insert(3).Delete(5), Create.Insert(5), unequivalent)`

are test cases for SET_OF_INT.

To execute test cases, we build a driver class which instantiates two objects O_1 and O_2 of class C. For each test case, the driver sends the first sequence of messages to O_1 and the second sequence to O_2, invokes a user-supplied operation, EQN, to check whether O_1 and O_2 are in the same state, and compares the result of this with the tag. We will discuss the EQN operation shortly. If, according to the specification, the two sequences should be equivalent, but EQN returns false, then an error is reported. Similarly, If the specification says the sequences should be unequivalent, but EQN returns true, an error is reported.

Notice that in this model, exhaustive testing corresponds to sending all possible pairs of sequences (perhaps using transitivity of equivalence to eliminate redundancies). If a class is exhaustively tested without exposing an error, then the class is guaranteed to be correct, assuming that the EQN operation is correct. This is the analog of the fact that exhaustive testing of a traditional program guarantees correctness, provided that no mistakes are made in checking whether the outputs meet the specification. Of course, as in the case of traditional program testing, exhaustive testing is completely impractical in all but the most trivial situations.

We now discuss the EQN operation. Ideally, the EQN operation in class C should check whether two objects O_1 and O_2 of class C are observationally equivalent, that is, it should check whether any sequence of messages ending in an observer yields the same result when sent to O_1 as when sent to O_2. Since it is clearly impossible to send every such message to the objects, in practice EQN will approximate a check for observational equivalence.

It is often quite easy to produce a recursive version of EQN from the specification of the ADT which C is intended to implement. For example a specification of a queue ADT gives rise to the EQN function,

```
EQN(q1,q2):
    if Empty(q1) then
        if Empty(q2) then return(TRUE)
        else return (FALSE)
    else if Empty(q2) then return(FALSE)
    else return(EQN(q1.Remove, q2.Remove))
```

which can easily be translated into the implementation language. Note that this is actually only an approximation of true observational equivalence because it neglects the possible effects of "building up" `q1` and `q2`, then removing elements. Thus, it might say that two objects are equivalent when they are not. Also, since EQN calls Empty and Remove, an error in one of these operations may propagate to EQN, causing it to "mask out" detection of the error.

Another approach to developing the EQN function is to write it at the "implementation level". In this approach, EQN is based on detailed knowledge of how data is represented and manipulated in the class body. For example, knowing that the queue is represented as a linked list, one can traverse the two lists comparing the elements. If sufficient attention is paid to the details of the representation, EQN can implement observational equivalence exactly. On the other hand, it is possible that the same misconceptions which lead to implementation errors in C's other methods may lead to errors in EQN.

It is also sometimes possible to use a very coarse approximation of observational equivalence as the EQN function. For example, we might consider two queues to be equivalent if they have the same number of elements, or if they have the same front element. As in the case of our specification-level EQN function, this version of EQN may consider two unequivalent objects to be equivalent.

Naturally, using a coarser approximation of observational equivalence will lead to less accuracy in the test results. It is interesting to note that deficiencies in the EQN function manifest themselves differently for our two different kinds of test cases. Consider a test case of the form `(S1, S2, equivalent)`. Suppose that the resulting objects O_1 and O_2 are not observationally equivalent, but EQN returns true. Then an error has gone undetected. This is unfortunate, but it is in the nature of testing to live with the possibility that errors will be missed. On the other hand consider `(S1, S2, unequivalent)`. Suppose again the resulting objects O_1 and O_2 are not observationally equivalent, but EQN returns true. In this case, a spurious error report will be generated.

2.1 Example: Testing An Incorrect Implementation of SET_OF_INT

Consider again the SET_OF_INT ADT whose specification is shown in Figure 1. Figure 2 shows a buggy implementation of SET_OF_INT, in which a set is represented by a linked list, the insert operation inserts the new element at the front of the list, and the `Delete(x)` operation removes the first occurrence of `x`. The EQN(O_2) message, when sent to O_1, traverses the O_1's list checking that each element is a member of O_2, then traverses O_2's list checking that each element is a member of O_1.

This implementation is incorrect because either the `Insert(x)` operation should check that `x` is not already in the list, or the `Delete(x)` operation should remove all occurrences of `x`.

By reasoning about an informal specification, or by using the test generation technique described in section 3 below, we can generate the test case

```
(Create.Insert(1).Insert(1).Delete(1), Create, equivalent),
```

which says that the object obtained by creating a new set, inserting 1 twice, then deleting 1 should be in the same state as an object obtained by simply creating a new set. To execute the test case, we could use a driver like the one shown in Figure 3 which declares two variables O_1 and O_2 of class SET_OF_INT, sends the first sequence of messages to O_1, the second sequence to O_2, then sends EQN(O_2) to O_1. As illustrated in Figure 4, EQN returns false. Comparing this with the "equivalent" tag on the test case, we see that the error is exposed.

Of course, in practice we don't want to "hard-wire" a specific test case into the driver, as we have done in the above example. Instead, we can build a more sophisticated driver which reads in the test case from a file, sends the given sequences of messages to objects of the class under test, invokes EQN, and compares the result with the tag. Such a driver has very limited dependence on the specifics of the class under test. It is thus possible to automatically generate a test driver for class C from the interface of class C. In section 3 we describe a tool which generates such drivers, and an interactive tool for generating test cases from an algebraic specification.

3 Description of the Tools

In this section, we describe ASTOOT - **A** Set of **T**ools for **O**bject-Oriented **T**esting, a system for testing classes based on the concepts explained in section 2. ASTOOT contains three components: the driver-

```
      -- -- Buggy SET: ``delelete" contains a bug
      -- -- EQN uses ``inclusion" of sets to test equality
class SET export
   empty, insert, delete, member, eqn
inherit
   LINKED_LIST [INTEGER]
      rename
         present as member, delete as ll_delete
feature
      -- -- Insert v into current set
   insert (v:  INTEGER) is
      do
         start; insert_left(v)
      end; -- -- insert

      -- -- Delete only the first v in the current set
   delete (v:  INTEGER) is
      do
         search(v, 1);
         if not offright then ll_delete end;
      end; -- -- delete

      -- -- Two sets are equivalent if
      -- -- (Current includes s) and (s includes Current)
   eqn (s:  like current):  BOOLEAN is
      local stop:  BOOLEAN
      do
            -- -- Test whether s includes Current
         from start until offright or stop
         loop
            if s.member(value) then forth
            else stop := true
            end; -- -- if
         end; -- -- loop

            -- -- Test whether Current includes s
         from s.start until s.offright or stop
         loop
            if member(s.value) then s.forth
            else stop := true
            end; -- -- if
         end; -- -- loop

         Result := not stop
      end; -- -- eqn
end; -- -- Class SET
```

Figure 2: Buggy implementation of SET_OF_INT in Eiffel

```
var O1, O2: SET_OF_INT;

begin
    O1.Create;
    O1.Insert(1);
    O1.Insert(1);
    O1.Delete(1);

    O2.Create;

    equiv := O1.EQN(O2);
    if not equiv then
      Generate_Error_Report
end.
```

Figure 3: Hard-wired driver

```
      O1
Create
   empty list
Insert(1)
   → [1]
Insert(1)
   → [1] → [1]
Delete(1)
   → [1]

      O2
Create
   empty list

EQN returns false
```

Figure 4: State diagram

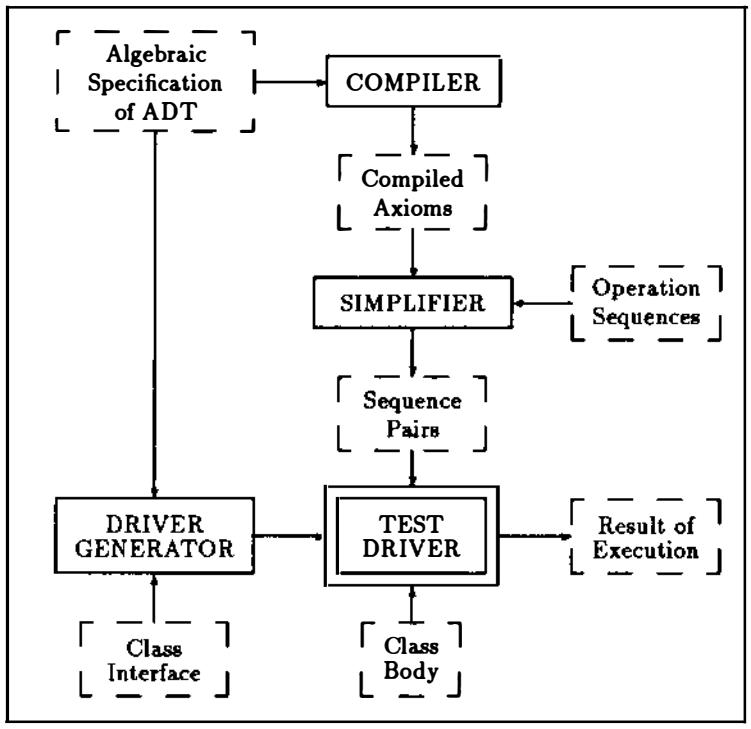


Figure 5: Components of ASTOOT

generator, the compiler, and the simplifier. The compiler and simplifier together form an interactive test generation tool, which generates test cases from an algebraic specification. The driver-generator automatically generates a test driver for a class from the class interface. This test driver can be used to execute automatically generated and/or manually generated test cases. The relationships between these components are illustrated in Figure 5.

3.1 The Driver Generator

When using the concept of sequence pairs to test a class, the test drivers become simpler. Furthermore, drivers for different classes are very similar to one another. This uniformity makes it possible to automate

the process of creating test drivers. The driver generator takes the class interface and syntactic part of the specification as input, checks whether both of them have the same exported operations, checks whether the implementation has an EQN operation with the correct declaration. For testing purposes, the operation EQN has to be exported by the implementation. If the implementation language has the facility of selective export (like Eiffel), then we can let EQN be exported only to the test driver, so the integrity of the implementation can be preserved. The driver generator then builds a test driver, which is a program in the same language as the class being tested. The current version of the driver generator is targetted to Eiffel.

The resulting test driver reads in a set of test cases, one at a time from a file. Each test case consists of two sequences of messages followed by a tag indicating whether they are supposed to be equivalent or unequivalent. Since the test cases may have been generated manually, the test driver contains a parser that parses the input operation sequences to make sure they are syntactically correct to decrease the possibility of human errors. For each operation sequence pair, the test driver declares two objects of the class under test, one for the *original sequence* (called *original object*), the other for the *simplified sequence* (called *simplified object*). The test driver then, reads in the first operation in the original sequence, makes sure it is a "create" operation, reads in the arguments of the operation as character strings, makes sure the arguments are in the proper forms for the corresponding types, converts the arguments into proper types, sets up necessary assignments for these arguments, and applies the operation with its arguments to the original object. This process is repeated for every operation in the original sequence. Then, the same process is applied for the simplified object. Next, the test driver reads in the tag for the sequence pair, uses EQN to compare the final states of these two objects, examines the tag, and reports an error if the result of EQN does not agree with the tag.

3.2 Test generation tools

The compiler and simplifier are used to generate test cases from an algebraic specification of an ADT. The compiler translates the axioms from LOBAS into a language-independent internal representation. The simplifier takes an initial sequence of operations, supplied by the user, translates it into an internal representation called an ADT tree, then uses the axioms as re-write rules to obtain equivalent sequences of operations. This process also suggests "interesting" unequivalent sequences. We now describe the compiler and simplifier in more detail.

3.2.1 Compiler

The compiler first reads in a specification written in LOBAS and does syntactic and semantic checking on the specification. Since LOBAS is a language with multiple inheritance and genericity, the semantic checking of the compiler is more complex than conventional compilers. In addition to conventional semantic checking, the compiler has to resolve inheritance and genericity constraints.

Under multiple inheritance, if a class has two ancestor classes having operations with the same name, the programmer must rename one of them. The compiler checks to make sure there is no name clashing between ancestors of the class. Also, the compiler has to check for circular inheritance, that is, a class being an ancestor of itself.

In LOBAS, a generic parameter may be put under constraint, as in the declaration of SORTED_LIST [T - > COMPARABLE]. When the compiler encounters a variable of type SORTED_LIST[NAME], it has to make sure the class NAME is a descendant class of COMPARABLE.

After syntactic and semantic checking, the compiler outputs a list of axioms, in the form of trees, which will be used by the simplifier as rewriting rules. For instance, the left-hand-side and right-hand side of axiom 6 in SET_OF_INT are shown in Figure 6.

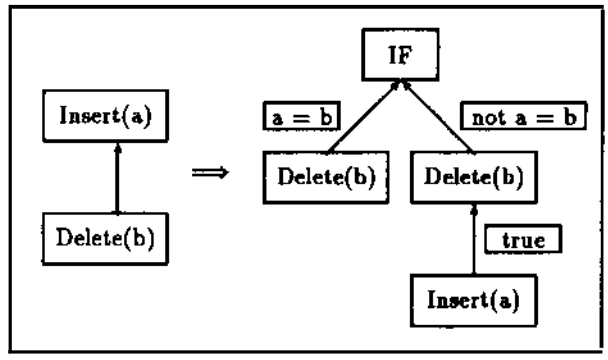


Figure 6: Axiom 6 of SET_OF_INT in ADT tree form

3.2.2 Simplifier

The simplifier takes a user-supplied sequence of operations and uses the rewriting rules generated by the compiler to obtain an equivalent sequence (i.e. a sequence which should put an object into the same state as the original sequence does).

Internally, the simplifier uses a tree called *ADT tree* to represent the states of an object. The branching of ADT tree arises from axioms having *IF_THEN_ELSE* expressions on the right hand side. Each node of the ADT tree is an operation with its arguments, which might also be ADT trees. Each edge of the ADT tree has a boolean expression, called the ***edge condition***, attached to it. Each path from the root to a leaf of the ADT tree represents a possible state of the ADT. A boolean expression, called ***path condition***, is the conjuction of all the edge conditions on that particular path. The path conditions in a given tree are mutually exclusive. The size of the tree may be exponential in the size of the initial sequence. To deal with this complexity, the simplifier can operate either in batch mode, which builds the entire tree, or in interactive mode, which allows the user to selectively guide the construction of a subtree.

In batch mode, the process of simplification is as follows:

- Search through the axioms to find an axiom with a left-hand-side that matches some partial path of the ADT tree,
- If an axiom is found, the simplifier will bind all the variables in the axiom to the proper arguments in the partial path of the ADT tree, and simplify the arguments. It then replaces the partial branch with the right-hand-side of the axiom.
- The process of matching-and-replacing stops when there is no matching axiom.

In order for the simplifier to work properly, the set of axioms in the specification must be ***convergent***, i.e., the axioms must have the properties of ***finite*** and ***unique termination*** [MUS80]. The property of finite termination ensures the process of simplification will not go into infinite loop. The property of unique termination makes sure that any two terminating sequences starting from the same operation sequence have the same final results, no matter what choice is made as to which axiom to rewrite or which axiom to apply first.

An example, involving batch-mode simplification of the sequence `Create.Insert(x).Delete(y)` for the SET_OF_INT abstract data type is shown in Figure 7.

The simplifier will generate test cases of the form:

`(Create.Insert(x).Delete(y), Create, equivalent)` with the path condition "x = y", and

`(Create.Insert(x).Delete(y), Create.insert(y), equivalent)` with the path condition "not x = y" for the ADT tree (3) in Figure 7.

In interactive mode, the user picks out one of the paths of the ADT tree to be followed, attaches one more operation to the branch, waits for the simplifier to simplify the ADT tree, then repeats the process, as illustrated in the appendix.

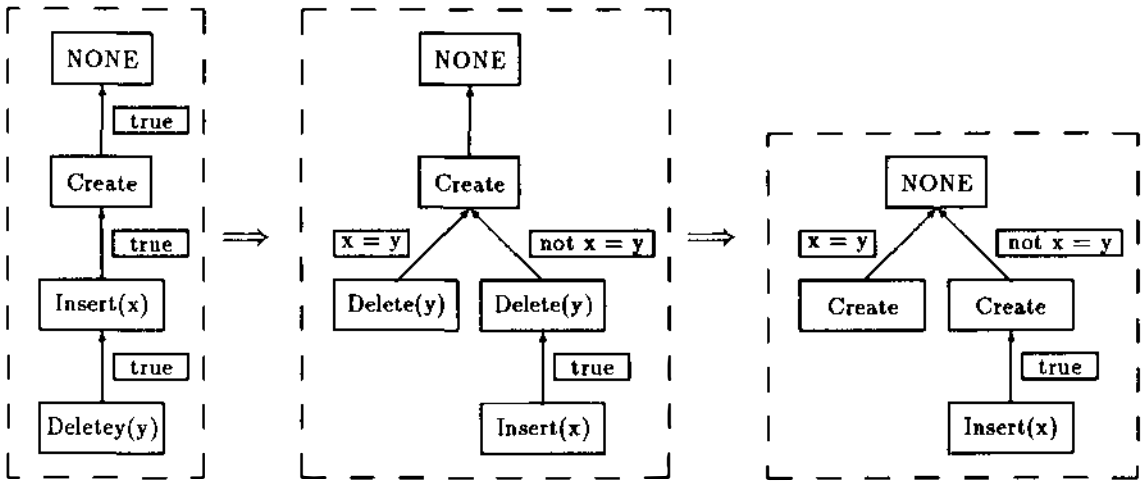


Figure 7: Simplification of create.insert(x).delete(y)

4 Inheritance

In this section, we discuss some of the ways in which inheritance impacts on testing. Suppose class C has been thoroughly tested and class C_d is a descendant of C. Under some circumstances, it is possible to "reuse" some of the results of testing C to reduce the effort of testing C_d.

We begin by noting that we can not expect the correctness of C_d to have any relation at all to the correctness of C unless there is a relationship between the specifications of the two classes. It is often, but not always, the case that the specification of C_d is a specialization or extension of the specification of C. The LOBAS specification language addresses this issue by incorporating a mechanism for inheritance of specifications. A descendent T_d of an ADT T contains all T's functions and may contain additional functions. Those functions which have been inherited from T obey the axioms of T. Additional axioms may further constrain the behavior of T_d.

Let T and T_d be the specifications of C and C_d respectively, where C_d is a descendent of C. Assume that C has already been thoroughly tested. In addition to testing C_d against it's own specification T_d, it is usually necessary to test C_d against T. This is because, given the dynamic binding mechanism, a variable of class C may be bound to an object of class C_d which uses the methods of C_d to respond to messages. The sender of such a message will expect the object to react to the message in accordance to the specification of C. In testing C_d against T, it may be unnecessary to perform a complete regression test (i.e. to re-execute all the tests used in testing C). This is because some of C_d's code has been inherited from C without modification. Thus, to check C_d against T it is only necessary to execute C_d those of C's test cases which involve methods which have been redefined.

Note that it is sometimes unnecessary for C_d to satisfy T. Programmers sometimes define a class C_d as a descendent of C strictly for convenience, in order to re-use C's code. For example, the implementation of SET_OF_INT shown in Figure 2 inherits from class LIST. A client of class SET_OF_INT (i.e. another class which sends messages to SET_OF_INT) may not care, or even be aware, that SET_OF_INT is implemented as a LIST, and thus may not care whether SET_OF_INT respects the specification of LIST. As long as none of the clients of C_d have the opportunity to assign an object of class C_d to a variable of class C, it is not necessary to check C_d against T. Of course, it is dangerous to omit testing of C_d against T unless the entire system has been carefully checked to assure that this is the case.

We next consider testing C_d against T_d. If T_d is a descendant of T some effort can be saved. Those test cases of T_d consisting of sequences of operations inherited from T will have already been considered in checking C_d against T. (As pointed out above, C_d may have been executed directly on them or the test

results may have been "inherited" from the testing of C, depending on whether they involved redefined methods). Thus testing C_d against T_d can focus on sequences involving operations which are unique to T_d and their interaction with operations of T.

Finally, we mention another way in which inheritance can affect test data selection. When testing C_d against T_d, it may be useful to consider "high-level" information about an ancestor C of C_d to help select instantiations of parameters in the operation sequences. Consider a class SET_OF_INT which is implemented as a descendent of the BINARY_SEARCH_TREE class. We would expect the order of insertions to be important, and might, for example want to include test cases such as

```
Create.Insert(w).Insert(x).Insert(y).Delete(z),
```

in which (w,x,y) represented different permutations of (1,2,3). On the other hand, if SET_OF_INT were implemented as a descendent of a LIST class, we would not expect the relative order in which items are inserted into a set to effect the results.

5 Conclusions

5.1 Related Work

We now compare our approach to related work on testing data abstractions. These systems generally fall into two categories: test execution tools and test generation tools, and are based on the traditional model of test cases. In contrast, our approach involves a new model of test cases, which we believe to be more natural for testing data abstractions, and which gives rise to test generation and test execution tools.

5.1.1 Test Execution Tools

One of the first systems to address the question of testing data abstractions was DAISTS (Data Abstraction Implementation Specification and Test System) [GAN81]. DAISTS uses the axioms of an algebraic specification to provide an oracle for testing implementations of the ADT. A test case is a tuple of arguments to the left-hand-side of an axiom. DAISTS executes a test case by giving it as input to the left-hand side and right-hand side of an axiom, then checks the output by invoking a user-supplied equality function (similar to our EQN).

Our test execution tool can be considered to be a generalization of DAISTS. For example, executing the DAISTS test case

```
(old = Create.Insert(1).Insert(2), a = 3, b = 2)
```

on the SET_OF_INT axiom 6 is equivalent to our test case

```
(Create.Insert(1).Insert(2).Insert(3).Delete(2), Create.Insert(1).Insert(3),
equivalent),
```

in which the second sequence is obtained by using axiom 6 to rewrite the first sequence.

However, DAISTS has no analog of our test cases of the form (S_1, S_2, **unequivalent**). This has significant ramifications: Even exhaustive testing with DAISTS may fail to detect an error which results in two states being erroneously "melded" into a single state. As an extreme example, consider an erroneous implementation in which none of the operations change the state of the object. The two sides of each axiom will return the same state on any input, and thus the error will not be detected.

A second distinction between DAISTS and our approach is that DAISTS requires the availability of a formal specification, while our test execution tool can be used when only an informal specification is available.

Another approach to testing ADT's has been proposed by Hoffman and Brealy [HOF89]. This approach facilitates test execution through the manual preparation of tables listing test cases along with their expected output. Each table entry consists of a trace (a sequence of operations putting the ADT into some state), exceptions raised by that trace, an observer, the type of the observer, and the expected value of applying the observer to that state. Tools to aid in generation and executions of these tables have been built. One distinction between this approach and ours is by using the EQN function to check outputs, in effect, we combine many of Hoffman's test cases into a single test case. It also appears that our approach leads to easier generation of test drivers. On the other hand, Hoffman's handling of exceptions is certainly an important idea, which we would like to try to incorporate into future versions of ASTOOT.

5.1.2 Test Case Generation

Two previous approaches to generating test cases from algebraic specifications have been reported. Gaudel and her colleagues [CHO86, GAU88] have developed a system in which the specification is written in a dialect of Prolog, a complexity measure is defined for sequences of operations, and the Prolog interpreter is used to generate sequences of operations of given complexities, sometimes subject to additional constraints. This approach might provide a useful means to generate interesting initial sequences for our simplifier.

Jalote et. al. [JAL88, JAL89] suggest that effective test cases can be generated from the syntactic part of an algebraic specification, without reference to the semantics. Experience with our tools indicates that in fact, it is very important to consider the semantic part as well, since different instantiations of arguments in a sequence of operations can lead to profoundly different states.

5.2 Future Work

The current version of ASTOOT is an experimental prototype. We note several possible enhancements. The operations in the test cases produced by the simplifier contain only symbolic arguments. It is left to the user to find an instantiation of the arguments which satisfies the path condition. In the future, we would like to add a constraint solver to substitute these symbolic arguments by values satisfying the path conditions. Then, we can feed the output of the simplifier directly to the test driver.

Although the compiler is written for the language LOBAS (Language for Object-Based Specification), this component can be replaced by a compiler which inputs specifications in another algebraic specification language and outputs ADT tree.

The concepts for driver generator can be applied to any implementation language with object-oriented features. The current version is targetted to implementations written in Eiffel. We would like to develop versions targetted to other object-oriented languages.

At the current time, the initial sequences for the simplifier is generated manually. In the future, we would like to develop an operation sequence generator which will generate operation sequences according to some testing criteria.

5.3 Summary

We have presented a new model of how to test classes which places emphasis on the fact that classes are implementations of data abstractions. In this model a test case is a pair of sequences of operations along with a tag indicating whether they should put objects into the same abstract state. A test case is executed by sending the sequences to two objects of the class under test, checking whether they are in the same state by using a user-supplied equivalence checking function, EQN, then comparing this result with the tag.

We then described ASTOOT, a set of tools based on this model. The driver generator automatically generates test drivers from the interface of the class under test. The compiler and simplifier partially automate the task of test generation, using an algebraic specification of the ADT under test as an input.

When no algebraic specification is available, the drivers produced by the driver generator can be used to execute manually generated tests.

Our initial experience using these tools has been encouraging. We have been able to efficiently generate and execute many more test cases than we would ever wish to generate manually. So far, we have used the tools to try to gain insight into how to generate effective tests.

One of the first things which has become apparent is that in classes whose specifications involve conditional axioms, the particular values used to instantiate parameters are important in test cases. For example we tested a heap implementation of a priority queue in which there was an off-by-one error in the sift operation. We executed 2771 test cases of the form

```
(Create.Insert(a).Insert(b).Insert(c).Insert(d).Insert(e).Remove,
Create.Insert(a).Insert(b).Insert(c).Insert(d).Remove.Insert(e), equivalent)
```

where `e` is not the largest element of `a,b,c,d,e`. Of them, only 280 exposed the error. This indicates that test generation methods which generate only one test case for each sequence of operations, without regard to the values of the parameters will not be very effective.

We plan to use ASTOOT for further experimentation, comparing the effectiveness of various testing strategies for object-oriented programs, and to incorporate our findings into future versions of the tools.

References

[CHO86] N. Choquet, "Test Data Generation Using a Prolog with Constraints", *Proceedings of the Workshop on Software Testing*, Banff, Canada, July, 1986, IEEE Computer Society Press.

[DOO90] R. Doong, "LOBAS Reference Manual," Polytechnic University Computer Science Technical Report, in preparation.

[GAN81] J. Gannon, P. McMullin, R. Hamlet, "Data-Abstraction Implementation, Specification, and Testing", *ACM Transactions on Languages and Systems*, 3, 3, July, 1981.

[GAU88] M.–C. Gaudel, B. Marre, "Generation of Test Data from Algebraic Specifications", *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, Canada, July, 1988, IEEE Computer Society Press.

[GOL83] A. Goldberg, D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass. 1983.

[GOG78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner, "An initial algebra approach to the specification, correctness, and implementation of abstract data types", in *Current Trends in Programming Methodology, v. 4*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[GUT77] J.V. Guttag, "Abstract data types and the development of data structures", *CACM* 20, 6, June 1977.

[HOF89] D. Hoffman, and C. Brealey, "Module test case generation," *Proc. of the ACM SIGSOFT '89 3rd Symposium on Software Testing, Analysis, and Verification*, Dec. 1989.

[JAL88] P. Jalote, M.G. Caballero, "Automated Testcase Generation for Data Abstraction," *Proc. COMPSAC 1988*, IEEE Computer Society Press.

[JAL89] P. Jalote, "Testing the completeness of specifications", *IEEE Transactions on Software Engineering*, SE-15, 5, May 1989.

[LIS75] B.H. Liskov, S.N. Zilles, "Specification Techniques for Data Abstractions," *IEEE Transactions on Software Engineering*, SE-1,1, March, 1975.

[MEY88] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[MUS80] D. R. Musser, "Abstract data type specification in the AFFIRM system," *IEEE Transaction on Software Engineering*, SE-6,1, Jan. 1980.

[STR86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.

[WEY82] E.J. Weyuker, "On Testing Non-testable Programs," *The Computer Journal*, 25,4, 1982.

APPENDIX

The following is a session of the simplifier in interactive mode.

```
Sequence --- create
Path(1) -- create

1) delete
2) insert
3) END
Which ----------------- (1 - 3) ? 2
Argument #1 ? a

Sequence --- create.insert(a)
Path(1) -- create.insert(a)
P.C. -----

1) delete
2) insert
3) END
Which ----------------- (1 - 3) ? 2
Argument #1 ? b

Sequence --- create.insert(a).insert(b)
Path(1) -- create.insert(a).insert(b)
P.C. -----

1) delete
2) insert
3) END
Which ----------------- (1 - 3) ? 1
Argument #1 ? c

Sequence --- create.insert(a).insert(b).
             delete(c)
Path(1) -- create
P.C. ----- c = b & c = a

Path(2) -- create.insert(a)
P.C. ----- c = b & ~ (c = a)

Path(3) -- create.insert(b)
P.C. ----- ~ (c = b) & c = a

Path(4) -- create.insert(a).insert(b)
P.C. ----- ~ (c = b) & ~ (c = a)

Select branch ........ (1 - 4) ? 4

Path(4) -- create.insert(a).insert(b)
P.C. ----- ~ (c = b) & ~ (c = a)

Is this ok ? y
1) delete
2) insert
3) END
Which ----------------- (1 - 3) ? 1
Argument #1 ? a

Sequence --- create.insert(a).insert(b).
 delete(c).delete(a)
Path(1) -- create
P.C. ----- a = b & ~ (c = b) & ~ (c = a)

Path(2) -- create.insert(b)
P.C. ----- ~ (a = b) & ~ (c = b) & ~ (c = a)

Select branch ........ (1 - 2) ? 2

Path(2) -- create.insert(b)
P.C. ----- ~ (a = b) & ~ (c = b) & ~ (c = a)

Is this ok ? y
1) delete
2) insert
3) END
Which ----------------- (1 - 3) ? 3
Save this Sequence ? y
N)ew sequence or Q)uit? q
```

The stored results indicate that create.insert(a).insert(b).delete(c).delete(a) is equivalent to create.insert(b), provided that a, b, and c are all distinct.

Developments in Fail-Safe Software

Robert L. Glass
Computing Trends
P.O. Box 213
State College PA 16804
USA (814) 234-0728

Abstract:

Removing all errors from software has proven to be beyond both the state of the practice and the state of the art. Yet some software, for example in life-critical systems, cannot be allowed to fail. What can be done?

Traditional error removal approaches (reviews, testing, proof) are necessary but not sufficient. Fault-tolerant techniques must supplement them. Fault tolerance involves diversity of both function and data. The key word here is diversity - redundancy approaches used by other engineering disciplines will not work for software. In this talk, the state of the art in fault-tolerant software approaches is presented.

The speaker:

Robert L. Glass is president of Computing Trends, a software engineering consulting and publishing firm. He has 30 years software experience in industry and five years experience as a professor in the Seattle University software engineering graduate program. His viewpoint is that of the experienced software practitioner, not that of a manager or academician. Published work relevant to this topic includes Software Reliability Guidebook, published by Prentice-Hall (1979), and "Some Thoughts on Software Errors," Notices of the German Computing Society Technical Interest Group on Fault-Tolerant Computing Systems, Dec. 1987.

Overview

Introduction

Why Fail-Safe [Fault Tolerant] Software?

Steps Toward Reliable Software

Steps Toward Fault Tolerant Software

State of the Art / State of the Practice

Information Sources

Summary

040788EDrlg1

Why Fault Tolerant Software?

"Good" software - 10-50 faults per 1000 LOC

Rigorous testing - 1-5 faults per 1000 LOC

"Software is the most complex task ever undertaken by humanity"

We simply don't know how to make software error-free

Steps Toward Reliable Software

Reviews - Management, technical; requirements, design, code, test, ...

Testing - More detail on next chart

Proofs - An area of controversy

040788EDrlg3

A Deeper Look at Testing

Requirements-driven (black box) testing

Structure-driven (white box) testing

Statistics-driven testing

Risk-driven testing

Phase-driven testing (unit, integration, system)

Steps Toward Fault-Tolerant Software

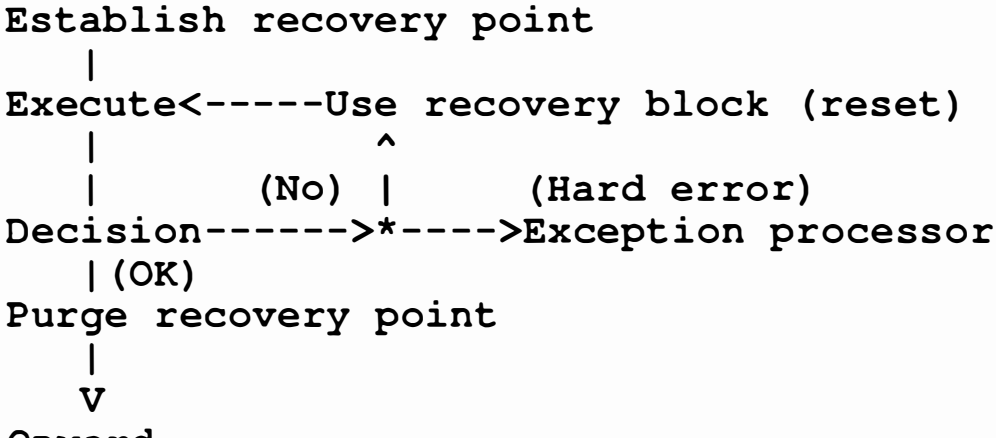
Principles
- Error confinement
- Error detection
- Error recovery
- Design Diversity

Methods
- Recovery blocks
- N-version programming
- Rollback/restart
- Data diversity

040788EDrlg5

Recovery Blocks

- Alternate section of code used if error detected
  - Recovery point - Acceptance test - Recovery block(s)



040788EDrlg6

N-version Programming

- Two or more complete versions
  - if > 2, "simultaneous" operation, vote, majority rules
  - if = 2,
    -- one operates 'til self-tests failure, then switches to other
    -- "simultaneous" operation, if disagree extrapolate prior results to pick winner (continuous functions only)
- Independent design/development
- Non-common faults(?)

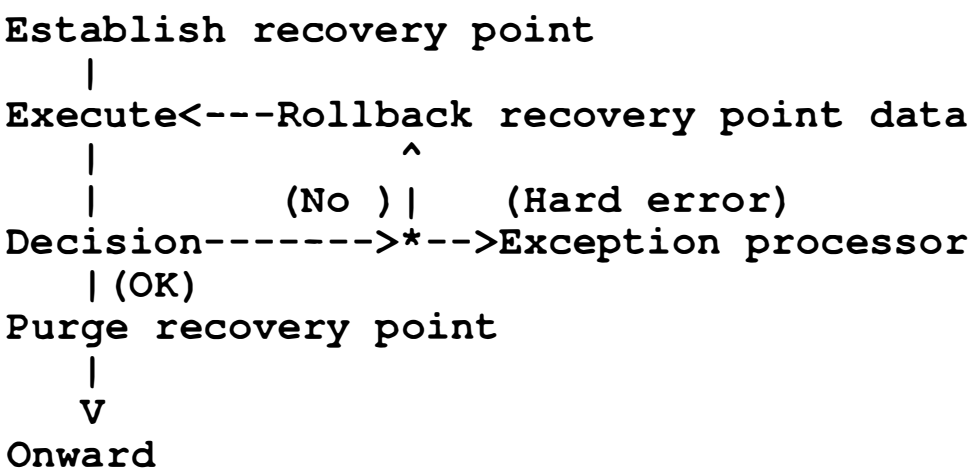
040788EDrlg7

N-Version Programming - 2

- Version 1---\
  Version 2----> Decision (majority rules)
  Version 3---/

Rollback/Restart

Tends to assume non-repeatable error



040788EDrlg9

Data Diversity

Redundancy added to data for self-check

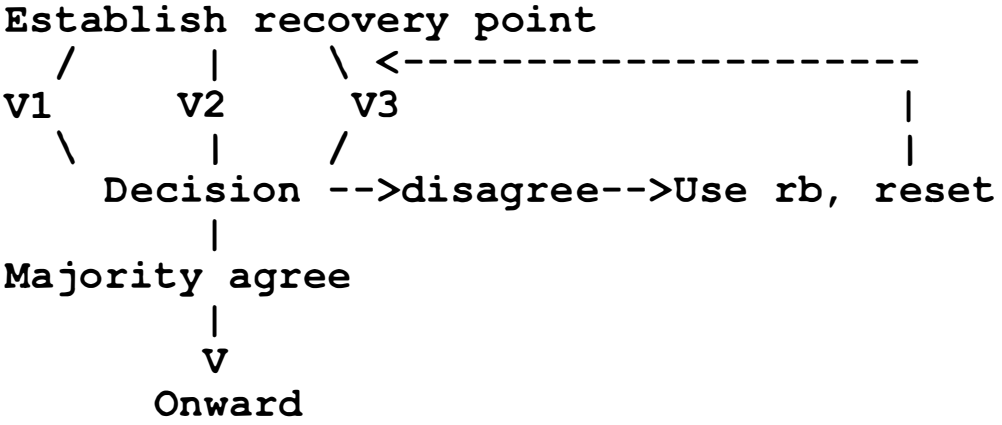
- Pointers (e.g., backward pointers in threaded list)
- Identifiers (unique to the node - access incorrect unless thru the node)
- Counts (e.g., count to check against terminal sentinel)

Audit systems typically as large as main logic

Combinations of the Above

Many combinations possible

"Consensus recovery block" = N-version + recovery block (rb)



040788EDrlg11

State of the Art / State of the Practice

The European connection

- More conferences, more findings published

The rail connection

The aerospace connection

Costs/benefits

Rail, Aerospace Connection

- Some practical examples:
  - Space shuttle [She78,Cau81,Gar81] (crew-assist)
  - Flight control for Boeing 737 et al, Airbus A-300 et al [Mar82,Rou86] (automatic recovery)
  - Railway interlocking control [Ste78,Hag86] (stop on fault detection)
  - (for refs see "Hardware and Software Fault Tolerance: Definition and Analysis of Architectural Solutions," Digest of 1987 CMU Symposium)
- Timing critical
  - Backwards recovery no good
  - Simultaneous operation no good unless parallel processors

040788EDrlg13

Cost of Fault Tolerance

Increases cost of * areas only
- Requirements (10%)
- Design * (10%)
- Code * (10%)
- Test *? (20%)
- Maintenance (50%)

Probably decreases cost of maintenance

Should strongly improve quality

Benefits of Fault Tolerance

Few examples in use

No quantitative answers

Intuitively, should work

N-version experiments cast doubts

Application dependent

040788EDrlg15

Information Sources

If it's more than 5 years old, it's probably out of date!

- Proceedings of the Third International Conference on Fault-Tolerant Computing Systems, Bremerhaven, Sept. 1987, Springer-Verlag
- Digest of Papers, the Seventeenth International Symposium on Fault-Tolerant Computing, CMU, July 1987, IEEE Computer Society Press
- Fault-Tolerant Computing - Theory and Techniques, editor = Dhiraj K. Pradhan, Prentice-Hall, 1986
- "Safety of Computer Control Systems 1986," (Proceedings of the Fifth IFAC Workshop, Sarlat, France, Oct. 1986), Pergamon Press

Rail-Specific Information Sources

- "ERICSSON Safety Systems for Railway Control," G. Hagelin, ERICSSON Document ENR/TB 6078, Oct. 86
- "Computerized Interlocking System - A Multidimensional Structure in the Pursuit of Safety," B.J. Sterner, IMechE Railway Engineer International, 1978, pages 29-30

040788EDrlg17

Summary

Fail-soft software must

- first be reliable (that won't be good enough)
- then have fail-safe additives (that is still more "researchy" than we would like)

Practical answers are at least as good as research answers at this point

INFORMATION SHARING DURING SOFTWARE DEVELOPMENT AND MAINTENANCE

Frank A. Cioch[1] **Fatma Mili**

Department of Computer Science and Engineering
Oakland University
Rochester, MI 48309
(313) 370-2200

Cioch@vela.acs.oakland.edu

Abstract

A framework is presented which outlines the types of information designers and programmers need to share during software development and maintenance. The purpose of the framework is to help designers and programmers determine what others need to know about their work, who needs to be told, when decisions should be made alone, and what information they should expect to receive from others.

The framework is based on the principle that the key to successful information sharing is an understanding of the identity and needs of the user community as well as the environment in which each software unit will be used. The user community for a given unit consists of designers and programmers who will develop software units which interact with that unit. The environment in which a given unit will be used consists of the uses to which that unit will be put.

Experiences using the framework in the private sector are also described.

Biographical Sketches

Frank Cioch is an assistant professor at Oakland University, where he teaches courses in software quality and software engineering. His research interests are the impact of design methods on structural quality and evaluating and measuring software understandability, maintainability and reusability. He received a Ph.D. in computer and communication sciences from the University of Michigan.

Fatma Mili is an assistant professor at Oakland University. Her research interests are in theoretical foundations of computer science and on decision support systems. She has been in technical and organizational committees of international conferences in both areas. She obtained a masters and a doctoral degree in computer science from the University of Paris, (P&M Curie), France.

---

1 correspondence author

INFORMATION SHARING DURING SOFTWARE DEVELOPMENT AND MAINTENANCE

Frank A. Cioch **Fatma Mili**

1. Introduction

The development and maintenance of a large software product necessarily involves the participation of many individuals, communication among whom is essential. Central to this process is the designer. An effective designer needs to be aware of the informational needs of others and to be able to provide pertinent information to the proper persons at the appropriate times. A designer is also an information recipient who must have both the ability to recognize when information is required and the ability to locate the source of the pertinent information.

One requirement for effective communication is being able to determine ***what*** information needs to be provided to others. To be effective, a designer must distinguish between information that is required by others and information that is nonessential and need not be shared with others at all. Both failure to impart necessary information and cluttering information provided with nonessential details hamper cooperative efforts. A designer must also be able to determine when product development will be facilitated by making decisions alone and when decisions need to be discussed with others. An understanding of what types of information are required by others will give designers a perspective from which to make this determination.

In addition to the content of the shared information, the timing of information sharing and the related issue of appropriate information recipients are important parts of effective communication. Because there are different points in the life cycle during which certain types of information are relevant to particular parties, a designer needs to be able to determine ***when*** to share information and ***who*** needs to be informed. The individuals interested in a given decision will vary, depending upon the timing within the software life-cycle , the position and responsibilities of the individual and the direct effects and possible implications of the decision.

In this paper a conceptual framework is presented which can be used by designers to determine what information needs to be shared with others and when and with whom to share it. It also can be used to determine the types of information with which a designer should expect to be provided. Specifically, a designer can use the framework to answer the following questions:

1 What do others need to know about my work?
2 Who needs to be told what?
3 Which decisions can I make alone and which should I discuss with others?
4 What information can I expect to be provided to me?

The information sharing issues will be discussed with respect to software units. The term unit is used to designate a cohesive portion of a program for which an individual designer is responsible. A unit may be thought of as a "module" or a "subsystem", but a unit need not be either a compilation unit or a system. Discussion will focus on the information needs of four types of individuals who are interested in a designer's decisions pertaining to a unit: (1) other designers, (2) implementation programmers, (3) maintenance programmers, and (4) management.

Other Designers

The information needs of other designers can best be understood by viewing those designers as users of the given unit. The designers/users not only want to know how to use the facilities provided by the given unit, but also what facilities are available and when it is appropriate to use candidate facilities. Because the unit is designed for use in a particular type of environment, in order to provide this information the user community needs to be informed of the characteristics of the environment in which the designer has assumed the unit will function. The user perspective can also be used by a designer to determine what information to seek from other designers.

Implementation Programmers

In order for an implementation programmer to code the unit, design documentation (like that described in software engineering textbooks [Pressman, 1987; Fairley, 1985; Sommerville, 1989]) must be provided. Although a design document gives a blueprint of what needs to be coded, an implementation programmer must still make coding decisions. Information must be provided which enables an implementation programmer to make coding decisions consistent with the goals and purposes which motivated the designer's choice of form and functionality of the unit.

Maintenance Programmers

Chapin [1988] has provided evidence that the maintenance life-cycle differs from development life-cycle in that maintenance requires start-up time to understand the existing program sufficiently to begin making the desired modifications. Start-up time exists not only for new personnel but also for developers assigned to new units during maintenance. In order to reduce start-up time the maintenance programmer must be provided with information which will facilitate learning how the unit to be maintained uses other units and how other units make use of it. Unlike the implementation programmer who must make coding decisions consistent with the factors motivating the unit's design, the maintenance programmer instead needs to understand how the environment in which the unit is used has changed over time and what modifications were required in the past to make the transition from the old to the new environment.

Management

Management needs to be kept up to date on progress made during development. In addition, management has a global interest in the program itself but need not be told specific details of each design or coding decision that is made. Management's basic concern is with the relationship between the program and the requirements that the program needs to satisfy. Thus, management needs to be informed about

decisions about each unit which pertain to requirements. Furthermore, if we assume that information needs of sales, marketing, and documentation take place through management, management will be interested in all decisions about the unit which impact end-users of the program. In addition, a technical manager desires an overview of how the various units in the program interact.

2. The Framework

The framework outlines information concerning the interaction between a given unit and designers and programmers who must develop and maintain units which interact with the unit. The set of possible uses of the unit is collectively referred to as the *environment* in which the unit will be used. Designers and programmers are viewed as *users* of the unit. The framework is based on the tenet that *the key to successful information sharing is an understanding of the user community and the environment in which the unit will be used.*

This environment is *not* the environment of the end-user but is the use that designers and programmers will make of the unit. The framework differs from a design in that it does not describe unit structure. Rather, it describes relationships between a unit and the users and uses of that unit. The framework also differs from a specification for the unit in that it is specifically concerned with user-unit and use-unit relationships whereas a specification is typically concerned with the unit per se rather than with the relationships between a unit and the environment in which a unit will be used.

Three relationships between a software unit and the environment in which the unit will be used are used to structure the framework:

1 What the unit does (external behavior or output considerations),
2 How the unit should be used (required setup or input considerations), and
3 When the unit should be used.

The framework also includes the rationale underlying unit-user and unit-use relationships. The provision of the rationale addresses management information needs, in that it includes relationships between the unit and both requirements and end-users.

To illustrate the framework, a unit called *Error Panels,* which is responsible for reporting unusual events to the end-user of a program by displaying an error panel on the screen, will be used as an example throughout this section.

2.1 What the Unit Does (External Behavior)

One way that a unit interacts with the user community and the environment is through the unit's external behavior. This includes everything that the unit does which impacts the environment. The external behavior of the unit has the following structure:

External Behavior

Function of the unit;	Rationale underlying the function.
Environmental effects;	Rationale underlying each effect.
Performance attributes;	Rationale underlying each attribute.

Function of the unit: Rationale underlying the function

A short verbal statement of the functionality provided by the unit is used to describe, from the user's point of view, the primary duty or role required of the unit. Users unfamiliar with the unit use this description to determine the purpose of the unit.

Designers and programmers who may want to inform an end-user about an error condition should be able to quickly tell from the description of the unit's function whether or not the unit is a candidate for the task. This can be accomplished by writing the function as a relation between the task performed by the unit and the use that should be made of the unit. In the error panels example, the task performed is displaying an error panel on the screen. The use that will be made of the unit is to report unusual events to the end-user. If the function section described only the task performed by the unit rather than a relation between task and use, the potential user would have to make the connection between the proposed use and the task.

Error Panels Function:

Report unusual events to the end-user by displaying an error panel on the screen.

Rationale:

Satisfy the requirement of uniform error handling and recovery in all modes of system operation.

The function is a relation between the task performed by the unit and the use that the users are expected to make of that unit. The rationale, an explanation of why this unit was identified during the design process, can provides information to management or to implementation and maintenance programmers. Some units exist because they satisfy requirements. For example, the error panels unit may exist because uniform error handling and recovery was promised. The rationale could instead provide an indication of the environment in which the unit was designed to operate. For example, the error panels unit may have been identified in the context of an object manager, where the panel is an object that must be manipulated by that object manager. If, at a later date, the object manager no longer exists, a unit to handle error panels may not be desired.

Environmental Effects: Rationale underlying each effect

The function of the unit determines its primary external behavior. This function is manifest through a set of individual effects on the environment. Each externally observable behavior of the unit should be noted by the designer and listed. For each effect, the rationale explains why the effect is produced. For example, the Error Panels unit may have the following list of environmental effects along with the accompanying rationale:

Environmental Effects:

1 Puts the passed message in the error panel.
2 It is selectable whether program execution continues or terminates.
3 Puts the passed message in an error log.
4 A keyboard event exists while the panel is visible.

Rationale:

1 Allows the user to tell the end-user what unusual event has occurred.
2 Necessary so that both fatal errors and warnings can be handled by the unit.
3 Ensures that all errors seen by the user are reported in the error log. This was an individual design decision.
4 Necessary so that a keystroke by an end-user can clear the panel.

The first three effects are behaviors produced by the unit while the fourth is a state of the environment that is produced as a result of invoking the unit. In order to interact with the unit appropriately, the users must be told what activities are performed for them by the unit. Furthermore, providing a list of environmental effects is one way the designer of the unit can distinguish between design decisions that can be made alone and those which need to be discussed with others. The user community needs to be informed of each design decision which results in an environmental effect.

Because implications of the environmental effect differ depending upon its source, the rationale underlying each environmental effect should be determined and noted. For example, an environmental effect which is the result of an individual design decision may be changed at maintenance time with little more than inconvenience to the user community. In contrast, change to an effect which is the result of carefully negotiated and planned cooperation between units during architectural design would ideally be implemented only after consultation with the parties to the original decision.

In the Error Panels example, an Error Log unit is also available so there must be some reason why the Error Panels unit puts the message in the error log rather than requiring that the user do this by using Error Log unit directly. If this was the result of negotiation during architectural design, it should be noted in the event that the agreed upon method is later questioned or a modification of it is sought. If the environmental effect was the result of a requirement, it should be noted as it is less likely to be negotiable.

An environmental effect resulting from the need to satisfy a requirement will be of interest to the technical manager, particularly if the decision transforms a relatively abstract requirement into a concrete externally observable behavior. Management will also be interested in environmental effects which impact the end-user of the program. In the error panels example, the rationale underlying each of the environmental effects will be of interest to management. For example, management would be interested to know that the reason that a keyboard event exists while an error panel is visible is so a keystroke by the end-user can clear the panel.

Performance Attributes: Rationale underlying each attribute

The external behavior of a unit includes not only actions resulting from its performance, but also includes characteristics of the performance itself, such as time

complexity or space requirements of the unit. For example, a unit which performs sorting may run in $O(n^2)$ or O(n log n) time. The value of using performance attributes during development is described by Gilb [1988].

The performance attributes of the unit influence whether or not it is appropriate for the environment in which the unit will be used. In the Error Panels example, a call to the Error Panels unit may result in a new heap allocation being performed. The users should be informed of this behavior because they must be aware that sufficient memory needs to be available at run time to perform a heap allocation. As with environmental effects, the rationale underlying each performance attribute should be determined and noted.

Performance Attribute:
Results in a new heap allocation being performed.
Rationale:
An error panel object is created by the object manager.

2.2 How To Use the Unit (Required Setup)

A second way in which the unit interacts with the user community and the environment is through the setup required for proper functioning of the unit. In addition to syntax considerations, this includes all actions prerequisite to proper functioning of the unit. The setup required of the user has the following structure:

Required Setup
Preconditions; Rationale underlying each precondition.
Use syntax; Rationale underlying use syntax.

Preconditions; Rationale
After determining the function of the unit, a potential user has a general sense as to what the unit does and can decide if the unit warrants further consideration. After examining the environmental effects and performance attributes a potential user knows in detail each of the behaviors produced by the unit. At this point, the user may know whether or not he or she wants to use the unit, but does not know if the unit *can* be used. The potential user must be informed of all of the prerequisites to proper functioning of the unit.

Each precondition that must be satisfied before the unit will function properly should be listed. For example, the Error Panels unit will not work unless the Error Log unit is available. Other preconditions may be related to the required run-time environment. For example, the Error Panels unit may have the precondition that enough run-time memory be available to perform a heap allocation. Knowledge of these preconditions will help users determine how to create the state required for proper unit functioning. As with external behaviors, this is required so that users can determine what activities they are responsible for performing and what activities are performed for them by the unit.

As with external behaviors, providing an explanation of why each precondition is required will be helpful to the designer of the unit in determining which decisions need to be discussed with other designers, to maintenance programmers in their decision making, and to management in relating the unit to requirements and end-users.

Preconditions:
1 Error Log unit must be available.
2 Enough run time memory available to perform a heap allocation.

Rationale:
1 The Error Panel unit uses the Error Log unit to add the error message to the error log.
2 If memory is not available to create the error panel object a fatal error will occur while your error is being handled. No memory will be available to produce an object for the resulting out of memory fatal error so the end-user will not know why the system went down.

Use Syntax; Rationale

Use syntax is directed toward programming considerations. Users need to know the type and purpose of each of the parameters. For example, the message that is to be displayed in the error panel must be formatted according to particular formatting characteristics. The names of required files must be provided along with how the unit is made operational. This type of information is desired by users who are already familiar with the unit. Such users already know that the unit should and can be used, but need to be reminded of the exact syntax of use and where required resources can be found.

Often rationale is not required for much of use syntax, particularly if the organization has calling sequence or development environment standards. However, it is useful for both users and maintenance programmers to know why certain parameters are required by the unit. Users can use this information to better understand how they are expected to interact with the unit. Maintenance programmers can use this information to better understand both how the unit is expected to be used and how the unit uses other units.

2.3 When To Use the Unit

External behavior and required setup will allow the potential user to decide whether the unit is applicable and whether the conditions are satisfied so that the unit can be used. However, analysis of external behavior and required setup often do not provide enough indication as to whether the unit *should* be used.

Users often need to be provided with a description of characteristics of the environment that affect either the optimal or intended use of the unit. Typically, the users of a unit are expected to use the unit under certain circumstances and an alternative unit in different circumstances.

When to Use:

Only events that must be brought to the end-user's attention should be displayed as an error panel. Problems from which recovery is possible and which require no corrective action by the end-user should be logged via the Error Log unit and should not be reported using Error Panels.

This information is required to promote consistency in all portions of the program. Otherwise, some users may be overzealous in using error panels while others may rarely, if ever, think an error panel is called for.

3. Use of the Framework

The framework is particularly well-suited to development environments in which the software product is structured as an integrated collection of application and support subsystems. For example, an information system designed to meet the information needs of executives may include application subsystems that provide the following capabilities to the executive: electronic mail, daily calendar, and data access to reports, charts and spreadsheets. Support subsystems, such as user interface, error panels and communications, which provide services to the application subsystems would likely be included in such an information system.

This type of development environment benefits from the framework in the following ways:

- Design and Implementation: The support subsystems are ideal candidates for unit documentation based on the framework. Since all developers of application subsystems must use support subsystems, their documentation can promote consistent use and reduce misuse. Furthermore, developers of the support units tend to be overburdened with questions from developers of the applications; the unit documentation can help reduce the need for disruptive informal communication,

- Maintenance: When a new application requires modification of an existing support subsystem, the unit documentation can help inform the user community of the proposed change. For example, an application subsystem that provides access to the Dow Jones News/Retrieval may require modification of the communications support subsystem and unit documentation could be used to inform users responsible for the electronic mail application,

- Implementation and Maintenance: When new personnel are added to the project, unit documentation for support subsystems will reduce start-up learning time by helping them learn how to write applications in the support environment.

The framework has been used in the private sector during software maintenance. The software, consisting of a set of application subsystems which used the services

of many support subsystems, contained approximately one-half million lines of code written in the programming language C. Many modules in application subsystems consisted primarily of calls to routines in support subsystems. Because of this, the architectural design of the product was difficult to describe. Structural diagrams quickly became too cluttered and hierarchical descriptions were too bushy to be of use. As a result, attempts at architectural design documentation had not been successful. The dozen designer/maintenance programmers communicated with each other only on an informal basis. The framework proposed in this paper was used to provide structure to the communication required for the maintenance effort by providing an agenda for weekly meetings at which support unit issues were discussed.

The first support unit selected for discussion was the object manager upon which all application units depended. Documentation based on the framework was developed for units comprising the object manager and were presented for group discussion. User misconceptions revealed at the weekly meetings motivated the designer of the object manager uncover potential errors resulting from unimagined misuse and to think of better ways to do things. The run-time memory precondition described in the error panels example given previously was an oversight that surfaced at one of these meetings.

Another problem revealed at the meetings was that many application programmers were writing code to perform functions that were performed by the object manager. The reasons were threefold: (1) lack of knowledge of the existence of the facility; (2) knowledge of facility existence but inability to readily learn how to use it; and (3) the facility did not do exactly what was required and a request for modification seemed more difficult and time consuming than writing code to perform the desired function. The unit documentation can help with the first two problems, particularly when one begins to view the system in terms of application and support units and users search for candidate facilities in the support units. Unit documentation could also help resolve the third problem were users consulted during definition of the support units rather than later in the life-cycle.

The biggest difficulty encountered in using the framework was that management required it to demonstrate usefulness without allowing for the development of tools to support it. As a result, the framework was directed toward stable features of the units. Useful information that was dynamic and required a tool to keep up to date, such as a cross reference of units used by a unit, were omitted. Despite this limitation, the framework proved useful as a communication vehicle and is currently being used as a standard for the types of information that designers/programmers are responsible for providing to others and, in turn, can be expected to be provided with.

The framework has also been used in an academic setting in software engineering classes [Cioch and Mili, 1990]. A term project problem was chosen that could be decomposed into a main unit and 3-5 interacting subunits. The entire group was responsible for the main unit and each group member was responsible for a subunit. The instructor played the role of technical manager. Each student was required to produce documentation for his or her unit using the framework, which was then used

as a communication tool during design. Experience obtained in the technical manager role suggests that the framework not only fostered communication and decision making during design but also proved useful in teaching students how architectural design differs from detailed design.

The proposed framework is not tied to any particular language or methodology. Although documentation containing information given in the framework can be developed without supporting software tools, an automated tool would make the effort more feasible. Given the network-like quality of the information in the framework, hypertext is being pursued as a way of implementing the framework described in this paper. A tool written in hypertext which can be used for communicating and organizing informal information has been developed at Microelectronics and Computer Technology Corporation (MCC) [Conklin and Begeman, 1988].

The versatility and stability of the framework is also being investigated by using it in a significantly different type of environment: the definition and design of a robotic vehicle for military utilization. Finally, use of the framework to document an entire system is being pursued. Were the framework to be used for documenting a complete system, in addition to each of the unit documents, a system-level document would have to be constructed as a way of managing and organizing information access. Automatic extraction, indexing, and cross referencing of information would be desirable.

4. References

[1] Pressman, R.S., *Software Engineering: A Practitioner's Approach*, 2nd ed., McGraw-Hill, 1987.

[2] Fairley, R., *Software Engineering Concepts*, McGraw-Hill, 1985.

[3] Sommerville, I., *Software Engineering*, 3rd ed., Addison-Wesley, 1989.

[4] Chapin. N., "Software Maintenance Life Cycle," *Proceedings of the Conference on Software Maintenance*, IEEE, 1988, pp. 6-13.

[5] Gilb, T., *Principles of Software Engineering Management*, Addison-Wesley, 1988.

[6] Cioch, F.A. and F. Mili, "Use-Perspective Unit Documentation," Proceedings of the 4th SEI Conference on Software Engineering Education, *Lecture Notes in Computer Science,* No. 423, Springer-Verlag,1990, pp. 136-144.

[7] Conklin, J., and M. Begeman, "gIBIS: A Tool for all Reasons," MCC Technical Report Number STP-252-88, Austin, Texas, July 1988.

Automated Adaptation of Programs

Amitava Datta and Prabhaker Mateti
Department of Computer Science and Engineering,
Wright State University, Dayton, Ohio 45435.
Email: adatta@cs.wright.edu, pmateti@cs.wright.edu

Key Words: Software Reuse, Maintenance, Adaptation, Static Structure of Programs.

Abstract

Software adaptations play a very important role in software reuse. Much of todays adaptations are performed using primitive tools like the text editor. We present a method of software adaptation where a software engineer first describes what he/she wants in the revised source code in a formal language called ASL. A semi-automated system then carries out the actual adaptations on the program. We use a rule-based approach to determine the actual sequence of adaptation operations to perform.

Amitava Datta

Received a M.S. degree in Computer Science from Case Western Reserve University, Cleveland, in 1986. Currently pursuing a Ph.D. degree in Computer Science at Wright State University, Dayton. His research interests are software systems, software engineering, particularly in the areas of software reuse and formal specifications of software.

Prabhaker Mateti

Ph. D. — University of Illinois at Urbana-Champaign, 1976
Associate Professor, Wright State University, Dayton, Oh 45435

His research interests are design environments, software engineering, programming methodology, and language design. He is currently leading an effort to build a *design environment*, called DÔME. Designs of software systems are expressed in a language, called ÔM , that gracefully merges functional programming concepts with predicative concepts. These designs are semi-automatically translated into efficient implementations.

Automated Adaptation of Programs

Amitava Datta and Prabhaker Mateti
Department of Computer Science and Engineering,
Wright State University, Dayton, Ohio 45435.

1 Introduction

Reusing portions of existing software source code in the construction of new software is both a cost-cutting and quality improving exercise. Towards this end, the software engineering community is developing software components analogous to hardware ICs and schemes of classifying such components [Freeman 87]. However, there are extensive repositories of well-written source code that have been in the past two decades gone without reuse considerations. In this paper, we address the question of how we can salvage pieces of reusable code from such repositories.

We view the salvage operation as consisting of (i) recognizing a piece as reusable, and (ii) adapting it into the required component. The former task requires "intelligent" browsing facilities provided by a system along with tremendous experience on the part of the human browser. In this paper, we concentrate on how the given piece of software may be adapted into the required component. It is in this activity that one must not get too carried away when thinking about reusability in a manner analogous to hardware ICs. Software source code is inherently a malleable medium. Disciplined adaptation of source code packages yields significant benefits of not only efficiency and customization, but also of unforeseen reuse.

Much of todays adaptations are performed using primitive tools like the text editor. Every step is performed manually by invoking character/line oriented commands. A small degree of automation in the adaptation process is accomplished via the use of "scripts" (like `patch` and `sed` scripts in the Unix environment). These tools view a program as a sequence of characters/lines. Text pattern matching is the primary mechansim that the tools use to locate places of change. Program transformation systems have been proposed to assist software reuse [Cheatham 83, Boyle and Muralidharan 83] but have found limited success. Section 8 discusses the contributions in this area and their limitations in supporting software adaptation.

In this paper we present a formal method to carry out software adaptations. We present the use of an adaptation specification language called ASL which we have designed to formally specify adaptations. We give an overview of a system called SSA to semi-automatically carry out the specified adaptation.

2 Our Approach to Adaptation

We view software adaptation as a highly technical activity on a par with designing and implementing new software. In our encounters, it is the more mature, and more able software engineer who is both comfortable and competent to perform adaptations. The newcomer prefers to write anew. However, analysis of the activity of adaptation shows that much of it is tedious and routine, occasionally requiring insightful changes. Our method caters to these observations. The engineer

describes what he/she wants in the revised source code; our system carries out the tedious parts, asking for intervention occasionally.

An adaptation specification is a formal expression which unambiguously states "what" is desired of the adapted program without stating "how" the adaptation needs to be performed. Given an adaptation specification we must then proceed to determine the actual steps required to obtain the desired adapted program. The separation of the specification and the "implementation" of the adaptation gives more clarity to the process of adaptation leading to more reliable adaptations.

We have designed a formal language called ASL to specify adaptations. The use of a formal language enables us to semi-automatically perform the actual adaptations. We use a rule-based approach to "executing" ASL specifications.

3 Scope of Our Work

We do not claim to handle all kinds of adaptation specifications that may be required on a given piece of software. In this section we establish the scope of our work. We are not going to dwell on what "adaptation" means, but will merely depend on the readers intuition.

3.1 Program Space

A program is a point in a n-dimensional program space. Each axis provides a specification of one "aspect" of the program. An axis can specify the functionality, memory usage, computation time required, file system usage, its module structure, etc. No one axis can completely describe all the aspects of a program. A point on any such axis is specified by an object describing one aspect of the program – presented, perhaps, in some formal language; the formal nature of the description just helps in accurately identifying a point in the different axes. Thus a functional specification of a program, in some formal language, identifies a point along the functional axis of this n-dimensional space. Similarly, other forms of formal descriptions could be used to identify a point in their respective axes of a program.

An adaptation can be specified as a change in some "aspect" of the given program. Ideally, adaptation specifications should be able to specify the change with respect to any axis in the program space. However, we find that specifications of change in the "run-time" behavior is, in general, hard to specify and verify in a given program. In the work presented here we restrict the adaptation specification to deal with aspects of a program that can be "statically" verified.

3.2 Static Structure

We identify *static structure* of a program as one of the axes in the n-dimensional program space. By *static structure* we mean the class of properties of programs that can be derived only by examining their parse tree. Call graphs, module interconnection, interface definition, data definitions, the use and update of variables, data dependencies, flow graphs etc., are all examples of the static structure of programs. These properties can be determined and verified without running the program. Any "run-time" property of a program, e.g., memory usage, average length of dynamically allocated lists, cpu usage, etc., are not considered as part of the static structure of the program.

We find that a large number of adaptations can be expressed as a change in their static structure. Even programs which go through elaborate functional modifications require a number of such adaptations. We design ASL to express changes of static structure of programs.

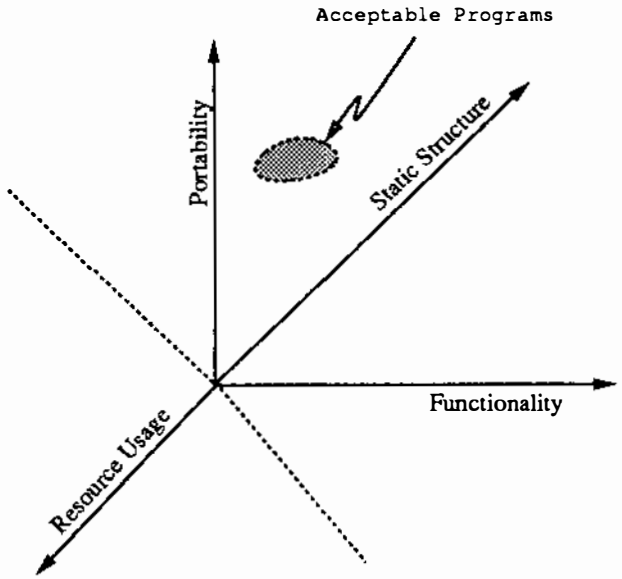


Figure 1: An n-dimensional Program Space: The shaded volume represents the region of "acceptable" programs which satisfy all the specifications.

4 ASL — An Adaptation Specification Language

The class of adaptations under consideration are indeed adaptations to the parse trees of the programs to obtain new parse trees. Any information about the structure of a program is derivable from its parse tree. In specifying and performing adaptations we find that *abstract syntax trees* (AST)[Aho and Ullman 78], a variant of the parse trees, provides a sufficient and compact representation of programs.

In this section we present the representation of programs that ASL is based on. We currently restrict ourselves to programs written in the class of procedural languages which includes Ada, Modula, Pascal and C. We present a brief overview of ASL. The following section presents example adaptation specifications in ASL. Execution of ASL specifications is presented in section 6.

4.1 Representation of Programs

ASL is based on the AST representation of programs. We find that by using such a representation for programs far more sophisticated adaptations are expressible than those possible using the sequence of characters/lines view used by text editors and command scripts. Figure 2 shows the AST of the following C function.

```
int g(int x, char c)
{
    int i;

    if(x > 10)
        i = 10;
    else
        i = 0;
    return x * i;
}
```

Every node in the AST represents some syntactic unit of the program. The circles indicate *simple* nodes whereas ellipses specify *complex* nodes. Intuitively, complex nodes denote syntactic units which are sequences of syntactic units of the program, e.g., a list of modules, declarations and statements. Nodes "inside" a complex node represent the *components* of the complex node. Thus a node representing a module declaration would be considered a component of some complex node representing the list of declarations in the module. Simple nodes, on the other hand, have their standard meaning as presented in [Aho and Ullman 78]. Both child and component nodes are ordered from left to right.

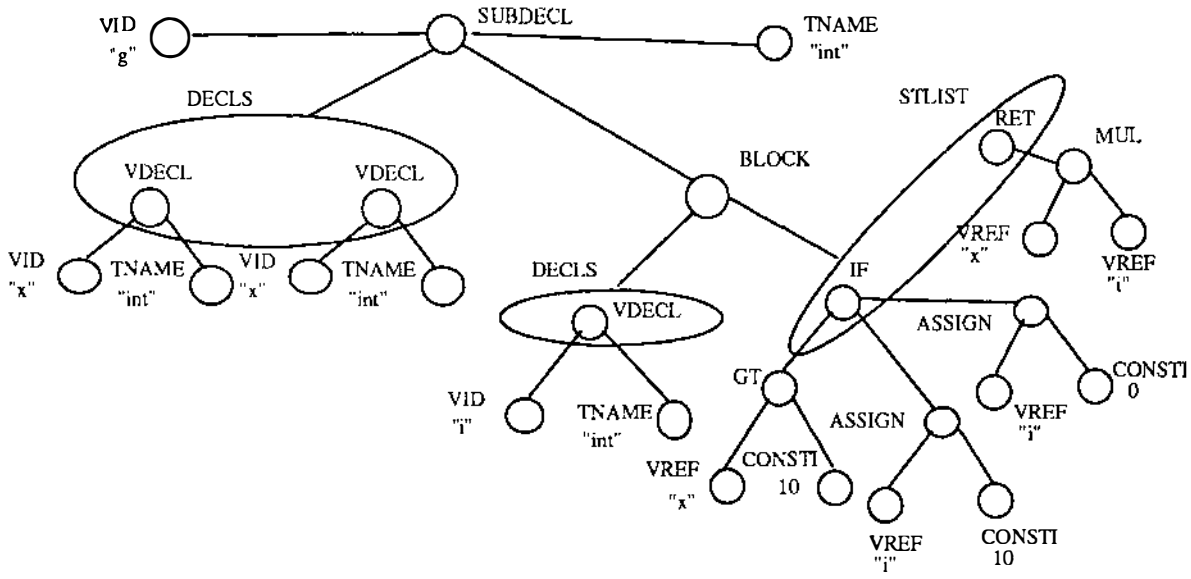


Figure 2: Abstract Syntax Tree of g()

With every node we associate an attribute called *kind*. In the figure above, labels next to the nodes indicate its kind. With simple leaf nodes we also associate a *value* attribute. For example, a node of kind `VID` (a variable id node) has a program identifier as its value.

All programs are modelled as a sequence of modules. Thus the root of the AST of a program is always a complex node.

4.2 An Overview of ASL

ASL is a specification language based on ÔM — a language used to specify the functionality of programs [Mateti 90]. ÔM is a language that includes higher-order logic and functions. It provides primitive data types such as sets, sequences, bags, tuples and tables. ASL extends ÔM by providing it the domain of ASTs. The language does not make any interpretation of the different "kind" of nodes. This makes ASL independent of any of the programming languages under consideration.

Any value in ASL is an "object". `5` is an object in ASL and is an instance of the set of natural numbers `nat`. A node in an AST is an object. An object can be named. For example,

```
object maxlen is 5;
```

defines the object `maxlen` as denoting the value 5.

ASL variables are typed. The primitive types are `node` and `tree`. Types are treated as sets of values. New types are definable by the user. The following type definition introduces `subroutine` as a new type in an ASL specification. It is the set of all nodes of kind `SUBDECL`.

```
type subroutine is node
    such that
        kind.node = SUBDECL;
```

All variables must be declared in the scope that they are used. The following declares `f` as a variable of type `subroutine`; that is, the value of `f` will always be an AST representing the tree of a subroutine.

```
f : subroutine;
```

ASL inherits all the primitve functions on sets, sequences, bags, tuples and tables from ÔM . ASL provides a rich set of functions on trees and nodes. New functions can be defined. The following is a simple example of a user defined function. This defines `cond` as a function which takes a node `n` of kind `IF` as an argument and returns a node which is the first child of `n` — the condition expression node.

```
function cond (n : node such that kind.n = IF) node is
    children.n [1];
```

Definitions like the above are provided only to provide a meaningful way to access the various syntactic units of the program. More complex functions can be defined. The following defines the function `encl` which returns the "innermost" block (a subroutine, the body of a subroutine, a complex statement or a module) in which the specified node `n` is found.

```
-- the innermost block in which n is found
function encl (n : node) block is
    if parent.n in (subroutine + block + module)
        => parent.n
    ::  => encl.parent.n
    fi
;
```

ASL functions are declarative in nature and are side-effect free, i.e. they do not modify any global state. In the examples above the result value of the functions is "directly" defined as the result of other primitive functions of ASL. A return value can also be implicitly defined by stating its property. In the following function the return value of `decln` is a declaration `d` from the set of declarations `ds` whose first child has a value equal to the specified identifier `ident`, i.e. a declaration of the identifier `ident`.

```
function decln(ident : identifier, ds : set decl) decl is
        d : ds such that
            value.(children.d [1]) = ident
;
```

The properties are specified using logical predicates. In the example above, the equality predicate '`=`' has been used. Both universal and existential quantifiers can be used in defining predicates.

5 Examples of ASL Specifications

In specifying the adapted program we need to formally state (i) all the properties (wih respect to their ASTs) that were true in the original program P and which are also true in the adapted program P', and (ii) properties that are "new" in P'; those that are not necessarily true in P. The latter may be specified as (a) properties stated just on the objects of P' and (b) as relationships between objects of P and those of P'. For any reasonable sized program, item (i) is typically going to be very large since the difference in P and P' is relatively small as compared to the size of the program. It would be inconvenient, if not impossible, to list all the static structure predicates of a given program that we would like to preserve in P'. It is for this reason an ASL specification only specifies item (ii). Item (i) is "implicitly" defined in all adaptation functions that perform software adaptation. Thus an ASL specification only specifies the "change".

Desired properties are stated as predicates on the ASTs of the original and the adapted program. All ASL specifications use the objects `T` and `T'` — the AST of the original and the adapted program, respectively. We define `R` and `R'` as the roots of the ASTs `T` and `T'`.

```
object R is root.T;
object R' is root.T';
```

Typically ASL specifications are specified using a number of auxiliary definitions. They include new type and function definitions. In all the examples presented below we only present the more important auxiliary functions. Others are explained informally.

Example 1. As a maintenance operation, it may be required to restrict the "size" of all subroutines in a given program, where the size of a subroutine is defined as the number of statements in it – including those that compose compound statements. At the specification stage it is only required to state the desired property. We do not attempt to outline the actual steps required to reduce the size of "long" subroutines.

In our specification, we first need to specify what we mean by the "size" of a subroutine. This following defines a function `size` which returns the number of statements that are defined in the body of a subroutine; this is what we would like to call the size of the subroutine.

`collect.(x, p)` is a user defined function which returns a sequence of those nodes in the breadth first traversal of the subtree rooted at node `x` which satisfies the predicate `p`. In its use below, the predicate is anonymous. `pred n : node is n in statement` defines a predicate which takes a node `n` as an argument and returns true if `n` is a statement node and false otherwise. `statement` is a user defined type and is defined as the set of all kinds of statements in the language.

`stlist.body.s` is the node representing the list of statements in the block defined by the subroutine `s` (`body.s`). The operator `#` returns the length of a sequence. In this case the sequence is that returned by `collect`.

```
function size(s : subroutine) nat is
    # collect.(stlist.body.s, pred n : node is n in statement)
;
```

We now define a function `issmall` which returns `true` if the specified subroutine has "size" less than the given `n`.

```
function issmall(s : subroutine, n : nat) boolean is
    size.s <= n
;
```

An adaptation specification is defined as a list of comma separated predicates — a conjunction of predicates. These specify the desired "properties" of the adapted program. Auxiliary definitions of types and functions appear after the keyword `adapt`. The properties follow the keywords `such that`.

In the following, `components.R'` returns the sequence of all `module` nodes that are part of the adapted program — components of the root `R'`. `subdecls` is a user defined function which returns the set of all subroutine declarations defined in the specified module. `semanticeq` is a user defined predicate over `T` and `T'` which holds if `T'` is derived from `T` using equivalence preserving transformations. The following specifies that the adapted program (with AST `T'`) is semantically equivalent to the original program (with AST `T`) and that for all modules `m` of the program the size all subroutines `s`, declared in the module, is less than 25 (`issmall.(s,25)`).

```
adapt

       :
<all auxiliary defns. go here>
       :

such that
    semanticeq.(T,T'),
    for all m : components.R'
        for all s : subdecls.m
            issmall.(s,25)

end.
```

Example 2. It is often required to remove direct accesses to global variables from the subroutines of a module. A number of different schemes can be used to perform this adaptation. However, at the specification stage we need only specify the desired property and not outline the actual adaptation steps.

We first define a function `access` which returns `true` if the variable with declaration `g` is global to subroutine `s` and is accessed by some statement of `s`. Otherwise it returns `false`. `vrefset` is a user defined function which returns the set of all variable references in a specified subroutine. `globals` is a user defined function which returns the set of all declarations that are global to the specified subroutine.

The function `access` returns `true` if there exists some reference to a variable global to `s` (`g in globals.body.s`) in the set of all variable references in the body of the subroutine `s` (`vrefset.s`) whose identifier (`vid.v`) is equal to that of the declaration `g` (`declid.g`).

```
-- variable references in subroutine
function vrefset(s : subroutine) set vref is
    collect.(stlist.body.s, pred n | node is n in vref)
|
function access (s : subroutine, g | vdecl) boolean is
    ( g in globals.body.s,
      exists v : vrefset.s
          such that
```

```
            vid.v = declid.g
    )
;
```

mvardecls.fh returns a sequence of global variable declarations in the module filehandler (defined as the object fh below). set mvardecls.fh returns the set of the elements in this sequence. The following ASL specification states that the adapted program is semantically equivalent to original program and that there is no direct access to the global variable buf in the subroutines of the module "filehandler" in the adapted program.

```
adapt

    :
-- The filehandler module in the adapted program
object fh : components.R'
    such that
      value.(children.m [1]) = "filehandler";

function noaccess(s : subroutine, g : vdecl) boolean is
    not access.(s,g);

such that
    semanticeq.(T,T'),
    for all s : subdecls.fh
        noaccess.(s, decln.("buf", set mvardecls.fh))
;
```

The adaptation above is specified on a module with the name "filehandler". If, on the other hand, the input program does not have such a module then the adaptation specification does not "suggest" any change to the input program.

Example 3. In a Unix environment, creation of a new process is requested via the call to the system call fork(). In certain situations a faster system call vfork() can be used. As a tune up operation, we would like to use vfork() instead of fork() whenever this situation holds. The actual adaptation performed in the example is a simple one — that of replacing the identifier fork() with vfork(). It is the capability of specifying where this can be done is of interest.

This example specification is applicable to programs containing calls to fork() which are in the form shown below.

```
if(fork() == 0)
{
  :
} else
{
  :
}
```

We first define a tree object which we can use as a "pattern" to "match" a call to fork(). We use the builtin function ast to generate the required pattern. It takes as argument a code segment in

the target programming language and returns its AST. Note that `ast` is target language dependent since it needs to know how to parse a given code segment. We use a node of kind **ANY** as a wild-card node which "matches" any node. **ANY** is thus a reserved node kind in ASL and represents the root of a subtree which matches any subtree. These nodes are identified using special '$' prefixed identifiers, e.g., `$1`, `$2`, etc. The following defines `X` as the pattern to use. The AST denoted by `X` is shown in figure 3.

```
object X is ast("if(fork() == 0) $1 else $2");
```

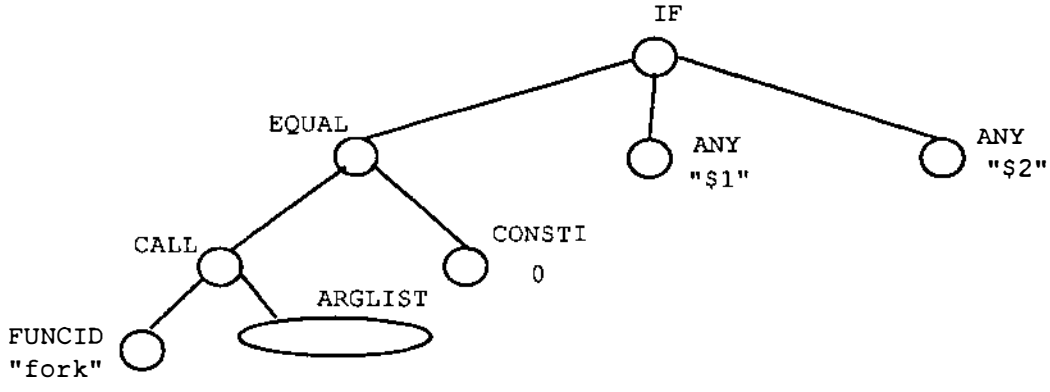


Figure 3: AST for `ast("if(fork() == 0 $1 else $2")`

`mod` and `use` are made available as user defined functions. Informally, `mod S`, for some sequence of statements `S`, is the set of identifiers of all variables which *may* get modified for a control path going through `S`. Similarly, `use S` is the set identifiers of all variables whose value *may* get used before any modifications to the variables due to the execution of a path going through `S`.

We define a function `mustusefork()` which returns true if all uses of `fork()` (of the form shown above) in the specified subroutine `s` are such that there exists some variable which is modified in the code executed in the child process (the 'then' part) and whose value is also used in the code for the parent process (the 'else' part). This indicates that `v` cannot be shared (as it is in `vfork()` and thus justifies the use of `fork()`. The operator `->` denotes logical implication.

```
function mustusefork(s : subroutine) boolean is
    for all n : set stlist.s
        match X st n ->
         exists v1 : vrefset.stlist.then.n, v2 : vrefset.stlist.else.n
            (vid.v1 in mod.stlist.then.n, vid.v2 in use.stlist.then.n,
             boundto.(s, vid.v1) = boundto.(s, vid.v2)
            )
;
```

`match X st n` returns true for any 'if' statement which has a condition expression `(fork()==0)`. The expressions `stlist.then.n` and `stlist.else.n` return the complex nodes representing the sequence of statements in the 'then' and the 'else' part of 'if' statement `n`, respectively. `boundto.(s,vid.v1)` returns the declaration to which the identifier `vid.v1` of the variable reference `v1` is statically bound to.

Given the above definition, we can specify the adaptation over all the modules of the adapted program P' as

```
adapt

    :
such that
    semanticeq.(T,T'),
    for all m : components.P'
         for all s : subdecls.m
             mustusefork.s
;
```

6 Executing Adaptation Specifications

We begin by making a copy of the subject program P. Let us call this P'. For all the predicates s in the given ASL specification S, we determine parts of P' that do not satisfy s. For all such parts we apply a sequence of adaptation operations on the AST of P' so as to satisfy s. Typically, this will not be a "one pass" process and will need to be done iteratively till all the predicates in S are found to be true in P'. The program P' is then the desired adapted program.

Note that actually making a separate copy may be a very expensive operation (with respect to both cpu time and storage required). Every node in an AST can be designed to have two sets of pointers — one relates to the edges in the AST of P and the other for that in P'. Both sets will have valid values only if the node appears in both the ASTs of P and P'.

The selection of a particular sequence of adaptation operations is done by the use of a rule-base. Adaptation rules can be defined by the user using a language called ADAPT. In the proposed adaptation system, a default set of such rules for commonly used adaptations will be provided. The rule base serves as a way to store "recipes" of adaptations that the designer performs given a particular ASL predicate to satisfy.

ADAPT rules specify an ASL predicate (by name) that it can "satisfy". We call this the *goal predicate*. A rule also specifies certain *applicability conditions* that must hold on the AST of P'. A rule gets "triggered" if the signature of the unsatisfied predicate s matches the goal predicate and the applicability conditions are found to be true. An adaptation rule specifies an *adaptation procedure* (the "action" of the rule) which can be applied whenever a rule is triggered. An adaptation procedure defines a sequence of adaptation operations. The application of an adaptation rule must guarantee the satisfaction of the predicate s.

An ADAPT Rule: The adaptation rule in this example is used to remove accesses to the global variable `g` in the subroutine `r`. There are a number of ways in which this can be done. This rule makes the variable `g` a parameter to the subroutine `r`. The goal predicate is the ASL predicate `noaccess`. The applicability conditions state that the variable `g` is a global variable, must not be directly modified in subroutine `r` and must be an atomic variable. Functions available in the ASL specifications are also made available for the specification of rules. This rule uses the following new types and auxiliary functions.

The adaptation procedure of the rule adds a new parameter declaration to `r`. It also modifies the argument list of all the call sites to `r` to have the new argument. The variable references in `r` remain unchanged.

```
rule noaccess(g : identifier, r : subroutine)
       [ boundto.(r,g) in globals.r, not g in dmod.r,
         is_atomic.(r,g) ] ->
```

```
{
    c : call_statement;
    n : node;
    t : tree;

    n := params.r;
    t := cons_c.(n, components.params.r | <boundto.(r,g)>);
    T' := replace.(T', n, t);

    for c in callsites.r
    do
       n := args.c;
       t := cons_c.(n, components.args.c | <root.cons_s.(VREF, g, <>)>);
       T' := replace.(T', n, t);
    od
}
```

As seen from the example above adaptation procedures use constructs — like assignment statements, conditionals, iterative statements and function/procedure calls, similar to those available in typical procedural languages.

More than one rule can be found to be applicable for a given unsatisfied instance of an ASL predicate. The default ordering of these rules is the order in which they are defined in the rule base. However, we find that the choice of a specific adaptation to be performed is often a matter of style, which differs from one user to another. It is for this reason we must provide means by which the default ordering can be changed by associating "priorities" with each rule. The priorities can be modified by the user as and when required.

Assuming that all the required adaptation rules are available, the process of adaptation terminates when all predicates in the adaptation specification are satisfied with respect to the adapted program (with AST T'). Our scheme does not make any attempt to detect unsatisfiable adaptation specifications. Often this may cause the adaptation system to execute in an infinite loop. It is left to the user to detect such a situation; perhaps by monitoring the adaptation steps — a facility provided by the adaptation environment discussed later.

7 The System for Software Adaptation

We are currently in the process of building the System for Software Adaptation (SSA) based on ASL and ADAPT. In this section we provide an outline of the architecture and the facilitities provided by the proposed system.

The following sections present the constituent modules of SSA. We also discuss the design of the internal form of programs in the system.

Target Language Processor

The Target Language Processor (TLP) is the only programming language dependent module in SSA. The proposed implementation provides a TLP which processes programs written in C. TLPs can be written for other languages as well.

The TLP is composed of two sub-modules — the language parser, and pretty printer. The parser is responsible for "loading" the subject program and representing it as an AST. Subject programs

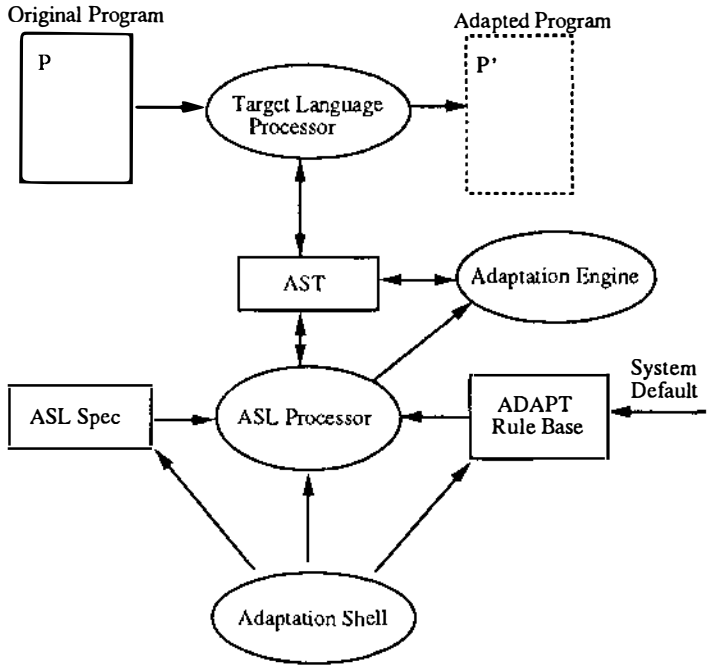


Figure 4: Architecture of SSA

must be syntactically correct. The pretty printer takes the AST of the program and produces the program in the target programming language.

The Adaptation Engine

This module performs the adaptation operations on the AST of the subject program. Adaptation commands are specified as a sequence of primitive adaptation operations organized as adaptation procedures presented in section 6. Determinig which adptation procedure needs to be applied is decided by the ASL processor based on the current state of the subject program and the ASL specification.

ASL Processor

The goal of the ASL Processor is to obtain an AST of the adapted program which meets the specified ASL specification. It determines what adaptation steps need to be carried out on the AST of the subject program. The actual adaptation is then performed by the Adaptation Engine.

The Adaptation Shell

The Adaptation Shell provides a convenient environment in which adaptations can be specified and carried out. The user directly interacts with this module. The following facilities will be provided.

1. A text editor to create/edit adaptation specification and enter/delete adaptation rules.
2. A code browser to browse through the current state of the subject program. Given the underlying "machinery" we should be able to provide "intelligent" browsing. To locate objects in the subject program one could define an object using facilities available in ASL. The browser can then scan the program to identify and display the desired objects.

3. A direct access to the adaptation engine. Even though this is not the preferred way to perform adaptations, this may be used for small "quick" adaptations. Adaptations performed in this manner do not come with any "guarantees".

4. A way to load the subject program and save adapted versions. This will essentially be the way to interact with the TLP.

5. A way to invoke the ASL processor. We feel that it will be necessary to provide a way to "step through" adaptations. This allows the user to monitor the adaptation process and abort it whenever the user detects any undesirable adaptations taking place.

6. A simple undo facility to revert back to the state before the last adaptation. (One invocation of the ASL processor will be considered as 'one' adaptation step).

8 Related Work

Software adaptations are essentially "source-to-source transformations". A large amount of work has been reported in this area. A detailed survey of well known transformation systems appears in [Partsch and Steinbruggen 83]. Such systems are guided by a set of *transformation rules* which specify "legal" transformations of "patterns" in the input program to generate the result program.

Transformation systems have typically been used for software development. The input to the system is a high level specification of the program. Repeated application of the transformation rules finally generate the result program which is the implementation of the high level specification. TAMPR [Boyle and Muralidharan 84], ZAP [Feather 82], the CIP system [Bauer et. al. 89], and the Harvard PDS [Cheatham 83] are some of the systems of this kind. In all such systems, the derivation of the implementation from a given specification can be saved. This forms an important document in capturing the design decisions that were made in the development process.

[Boyle and Muralidharan 83] and [Cheatham 83] propose the use of transformation systems in the process of reuse. Reuse is accomplished by the use of the same high level specification (also called *abstract programs*) in obtaining a number of implementations. An alternate sequence of transformations rules are used to obtain the new implementation which indicates a different set of design decisions. However, such specifications do not accompany the repositories of software that we wish to reuse. It is for this reason we cannot directly use such transformation systems the way it has been proposed. Obtaining specifications out of existing implementations (also called *reverse engineering*) is an ongoing research [Biggerstaff 89].

Transformation systems have also been used in the area of program "improvement". Program improvement relates to the activity of making an existing program more efficient and "tidier". [Loveman 76] and [Arsac 79] present catalogs of "meaning preserving" transformations implemented in a transformation system. The user of the system interactively invokes the transformations on a subject program. The Leeds Transformation System (LTS) [Maher and Sleeman 83] has similar goals. In this case, transformations are carried out automatically, guided by user provided transformation rules.

The transformation systems mentioned above require the transformation rules to be meaning preserving. We find that real life adaptations need not always preserve the meaning. GRAMPS [Cameron and Ito 84] is a *metaprogramming* scheme which supports arbitrary transformations. Metaprograms are programs written to manipulate other programs. The Pascal MPS, a system based on this scheme, provides a Pascal like language to express desired transformation. Programs are internally represented as parse trees. A number of specialized subroutines are provided to manipulate these parse trees. The facilities provided are similar to those provided by adaptation

procedures discussed in section 6. However, it does not provide any means of specifying adaptations as done in ASL. This makes it difficult to understand what a given metaprogram accomplishes.

The use of ASL differs considerably from the notations used in the systems above. We believe that in peforming adaptations we must be able to specify the desired adaptation without outlining how the transformations must be carried out. This is not the approach that current transformation systems use. We have used a declarative style which is very common in the domain of software specifications. Such a notation does not increase the expressive power but provides a more natural form of expression.

9 Conclusion

We have presented the concepts behind an adaptation language called ASL. ASL is a formal language which allows specifications to be declarative. We claim that declarative specifications are much easier to specify than the actual sequence of adaptation operations required to obtain the desired adaptation. ASL specifications are based on the AST representation of a program. Thus ASL predicates are statically verifiable. The following are some of the direct benefits of using our method of adaptation.

1. ASL specifications serve as documentation to record the adaptations performed on a piece of software.

2. ASL specifications can be written in such a way that they can be applied to more than one program. The possibility of reusing adaptations themselves is a very promising contribution.

3. When dealing with reasonably large sized programs, adaptations performed in SSA should take much less time than they would if done manually.

4. There will be a remarkable reduction of errors in the process of adaptation. We cannot rule out the possibility of errors arising from some unforeseen side-effects not handled by some adaptation rule.

We are currently building an interactive software adaptation system, SSA, based on ASL. We use a rule-based approach to making ASL specifications executable. The current implementation is targeted towards programs written in the C language.

References

[Aho and Ullman 78] Aho, A.V. and Ullman, J.D., *Principles of Compiler Design*, Addison Wesley, 1978.

[Arsac 79] Arsac, J.J., "Syntactic Source to Source Transforms and Program Manipulation," Communications of the ACM, January 1979, Vol. 22, No. 1, pp. 43–54.

[Bauer et. al. 89] Bauer, F.L., Moller, B., Partsch , Pepper, "Formal Program Construction by Transformations — Computer-Aided, Intuition-Guided Programming," IEEE Transactions on Software Engineering, Vol 15, No.2, Feb 1989

[Biggerstaff 89] Biggerstaff, T.J., "Design Recovery for Maintenance and Reuse," IEEE Computer, July 1989, pp 36–49.

[Boyle and Muralidharan 84] "Program Reusability through Program Transformation," IEEE Transactions on Software Engg., Sept. 1984, pp 574-588.

[Cameron and Ito 84] Cameron, R. D., and Ito, M. R., "Grammer-Based Definition of Metaprogramming Systems," ACM Transactions on Programming Languages and Systems, Jan 1984, Vol. 6, No. 1, pp. 20–54.

[Cheatham 83] Cheatham, T.E. Jr., "Reusability through Program Transformation," ITT Proceedings of the Workshop on Reusability in Programming, Sept 1983, pp. 122–128.

[Feather 82] Feather, M.S., "A System for Assiting Program Transformation," ACM Transactions on Programming Languages and Systems, Vol. 4, No. 1, Jan 1982, pp. 1–20.

[Freeman 87] Freeman, P., *Tutorial: Software Reusability*, The Computer Society of the IEEE, 1987.

[Loveman 76] Loveman, D.B., "Program Improvement by Source to Source Transformation," Third ACM Symposium on POPL, Jan. 1976, pp. 140–152.

[Maher and Sleeman 83] Maher, B. and Sleeman, D. H., "Automatic Program Improvement: Variable Usage Transformation," ACM Transactions on Programming Language and Systems, Vol. 5, No. 2, April 1983, pp 236-264.

[Mateti 90] Mateti, P., "ÔM : A Formal Language for Software Design and Specification," Tech. Report WSU-CS-90-7, 1990.

[Partsch and Steinbruggen 83] Partsch, H. and Steinbruggen, R., "Program Transformation Systems," Computing Surveys, Vol. 15, No. 3, Sept. 1983, pp. 199–236.

APPLYING SENSITIVITY ANALYSIS ESTIMATES TO A MINIMUM FAILURE PROBABILITY FOR SOFTWARE TESTING

Jeffrey M. Voas[1]
Systems Architecture Branch
Information Systems Division
NASA Langley
Hampton, VA 23665

Larry J. Morell[2]
Department of Computer Science
Hampton University
Hampton, VA 23665

Abstract

Sensitivity analysis is a new technique for monitoring the input/output behavior of a program. This paper describes a method of estimating the testing complexity of a program based upon the *sensitivity* of the program. From sensitivity analysis, a new testing complexity metric is derived as well as a scheme for determining where to concentrate limited testing resources.

Index Terms: testing complexity metric, sensitivity analysis, software testing, failure probability.

Dr. Voas obtained a bachelors degree in computer engineering from Tulane University in 1985 and a masters degree in computer science from the College of William and Mary in Virginia in 1986. Dr. Voas earned his Ph.D. in computer science from the College of William and Mary in Virginia in the spring of 1990. His current research interests are in ultra reliable software, studying propagation of data state errors, and structural analysis techniques for estimating and improving software reliability.

Dr. Morell's current research efforts are in fault-based testing, probable correctness, and error-flow testing. In the past he has investigated and published in the areas of fault-based testing, validation of expert systems, and specification of software tools. In addition, his curriculum module on Unit Testing and Analysis for the Software Engineering Institute is widely distributed and employed in both industry and academia. Dr. Morell obtained a bachelors degree in mathematics and computer science from Duke University in 1974 and a masters degree in computer science from Rutgers University in 1976. Dr. Morell earned his Ph.D. from the University of Maryland in 1983 with a thesis was in the area of of fault-based testing. He is an associate professor with the Department of Computer Science at Hampton University in Hampton Virginia.

[1] Author supported by National Research Council Associateship Program Grant NASW-3458 and NASA Grant NAG-1-884.
[2] Author supported by NASA Grant NAG-1-884.

1 Introduction

A program location is termed *sensitive* if it is difficult to "hide" a fault at that location. An example of a location is an assignment statement or "if" statement. A fault is considered difficult to hide at a particular location if the fault's inclusion at the location forces a discernable change to the input-output behavior of the program. If we can determine that each location in a program is sensitive, then we can determine the amount of testing needed for the program, as well as where to concentrate limited testing resources.

Consider the following analogy. If software faults were gold, then testing would be gold mining. Sensitivity analysis would be a geologist's survey done before mining takes place. It is not the geologist's job to dig for gold. Instead, the geologist establishes the likelihood that digging at a particular spot would be rewarding. A geologist might say, "This valley may or may not have gold, but if there is gold, it will be in the top 50 feet, and it will be all over this valley." At another location, the geologist might say, "Unless you find gold in the first 10 feet on this plateau, there is no gold. However, on the next plateau you will have to dig 100 feet before you can be sure there is no gold."

When software testing begins, such an initial survey has obvious advantages over testing blind. From this analogy, sensitivity analysis is our "geologist" in the software domain. Sensitivity analysis gives the testing intensity, whereas the geologist gives the digging depth. Sensitivity analysis provides the degree of difficulty which will be incurred during testing of a particular location to detect a fault. If after testing to the degree specified by sensitivity analysis only to find no failures, we then can feel confident that our location is *correct.*

Sensitivity analysis [2, 7] measures the fluctuation (or sensitivity) of the input/output behavior of a program to the presence of faults at various locations.[1] A method for analyzing sensitivity has been developed called PIA(for Propagation and Infection Analysis) [3, 2, 7]. Propagation and infection analysis employs aspects of both software testing methods and verification techniques [1]. The results of propagation and infection analysis distinguish it from traditional verification efforts however. Verification compares a program with its specification. The ultimate goal of verification is to show the program is correct with respect to its specification [4]. Propagation and infection analysis characterizes a program in terms of how its input/output behavior will be impacted by the presence of faults at various locations. A location which minimally impacts the input/output behavior is termed *insensitive.* The goal of propagation and infection analysis is to identify insensitive locations. Since program correctness, safety, and reliability are all intricately connected to the presence (or absence) of faults in the code, sensitivity analysis provides information useful in quantifying the effectiveness of other verification activities.

The *failure probability* of a program is the probability that an input point selected from an uniform distribution of input points results in failure. PIA simulates the way faults interact with the failure probability of the program. PIA answers the question "*If there were a fault*

[1] By "fault", we mean both the conventional syntactic fault as well as a data state error.

in this statement, what would be its minimal impact on the program's failure probability?" This "minimal impact" can be used in estimating the amount of program testing needed. If the "minimal impact" is small, the amount of testing needed is great. Thus we will use PIA to derive a quantity which we term a testing complexity metric. We define a *testing complexity metric* to be a measure of the testing required to offset the ability of a program to hide faults. The greater the testing complexity metric of a program, the easier it is for faults to be hidden and not revealed through testing.

To define what is meant by "impact" in the question above, a general model of sensitivity has been developed that relates faults to failure probabilities [2, 7]. The sensitivity analysis model can be used to estimate the minimal failure probability of the program [7]. We then estimate the amount of testing required for a particular confidence in the assertion that the program is "approximately" correct [5]. This estimate of the amount of testing required yields the following intuition upon which our testing complexity metric is based: *the amount of testing required to achieve a particular confidence in the assertion that the program is "approximately" correct is the most important measure of testing complexity.*

The remainder of our paper is organized as follows. Section 2 defines the notion of the minimal failure probability; this minimal failure probability estimate is necessary for the testing complexity metric, Υ, which is found in Section 3. Section 4 uses the estimate of the minimal failure probability along with the execution frequency of a program location as a guide for where to concentrate testing efforts.[2]

2 Latent failure rate

The maximum failure probability of a program with a particular fault is 1.0. It is, however, more useful to determine what the minimum impact on the failure probability will be for any fault. Although it will not be possible to estimate the minimum impact for "any" fault, it will be possible to estimate the minimum impact for a class of syntactic faults and a class of data state errors. Hence any fault that occurs in our classes should have a failure probability greater than or equal to our estimate. If we make the classes broad, then confidence in our process of finding the estimate is increased. And thus the confidence in our estimate also increases.

The *latent failure rate* is the quantity we use for achieving this. The *latent failure rate* of a location l is defined as follows: the minimum probability that if a fault exists at location l, and if location l is reached, then failure occurs. Note that the granularity of the term "location" in our geology analogy can change. For instance, the *latent failure rate* of a path equivalence class p is defined as the minimum probability that if a fault exists at a location on path equivalence class p, and if the path equivalence l is executed, then failure occurs.[3]

[2]The *execution frequency* of a particular location is the probability of executing the location under the input distribution used during infection analysis and propagation analysis.

[3]Path equivalence class defined in Appendix A.

And the *latent failure rate* of a program is defined as the minimum probability that if a fault exists in the program and if the fault is reached, failure occurs. In our analogy, this is the geologist telling us how much digging to do for various mines. The *latent failure rate estimate* is the estimate of the latent failure rate probability and is denoted by β_l. Formulae for finding the latent failure rate estimate are found in Appendix A. An intuitive discussion of what the latent failure rate estimate represents follows.

For a program to fail, three conditions must be met:

1. A fault must be reached,
2. The fault must adversely affect the computational state of the program, and
3. The altered program state must adversely affect the output.

The latent failure rate estimate attempts to quantify the minimal probability of failure occurring assuming the a location is reached. Since we assume that a location is reached, the latent failure rate estimate is just the probability that a fault at the location will adversely affect the location's computational state, and the probability that this altered state will affect the output. In loose terms, the latent failure rate is the probability that an executed fault will cause a failure.

Latent failure rate estimation uses statistical estimates of the frequency with which conditions (2) and (3) occur. We inject syntactic faults at a location and get an estimate of how frequently the computational state is altered by these faults. We also replace the current computational state of a program with an altered computational state, and see how frequently this replacement affects the output. We then select the minimum frequency with which condition (2) occurred, and the minimum frequency with which condition (3) occurred. Appendix A further explains how to use the minimum frequencies to determine latent failure rate estimates. We then select this latent failure rate estimate over all program locations, and get the program's minimum failure probability estimate. Hence we have an estimate of the minimal probability of failure of the program (if a fault exists and is reached) based upon the faults we inserted and the computational data state errors we inserted.

3 Applying the latent failure rate estimate to Υ

This section applies the path equivalence class latent failure rate estimate to a program testing complexity metric. Section 3 uses notation defined in Appendix A. It is also possible to determine a testing complexity metric for a single location or a single path equivalence class using the formulae of Appendix A. Formulae for a testing complexity metric for a single location or path equivalence class are not presented in this paper however.

If we jump back to our mining analogy, our testing complexity metric in the "software domain" is just the sum over all locations of the number of times the geologist tells us to

dig at a particular location in the "mining domain." In the ensuing discussion, it is assumed that all latent failure rate estimates are positive and non-zero. Our complexity metric, Υ, is based in part upon the work found in [6, 5]. In [5], Hamlet proposes the notion of "probable correctness" based upon the work of Valiant [6]. *Probable correctness* [5] attempts to predict from a successful test of the program whether the program is correct. Probable correctness is different than software reliability; software reliability attempts to predict the likelihood that the program fails in a given time interval during use.

Since β_x represents a point estimate of the minimal failure probability of location x, then $1-\beta_x$ is the probability of one successful execution involving location x, $(1-\beta_x)^{n_x}$ represents the probability of n_x successful executions of location x, and

$$1 - (1 - \beta_x)^{n_x} \tag{1}$$

represents the probability of at least one failure in n_x executions due to location x. Then

$$(1 - \beta_x)^{n_x} = 1 - \alpha \tag{2}$$

is the confidence that the true failure probability is not less than β_x. This is our confidence that β_x is a lower bound on the true failure probability. And

$$1 - (1 - \beta_x)^{n_x} = \alpha \tag{3}$$

is the confidence that the true failure probability is less than β_x. So when $\beta_x \approx 0.0$, α is the confidence that the program is "approximately" correct. Recall our testing complexity intuition: *the amount of testing required to achieve a particular confidence in the assertion that the program is "approximately" correct is the most important parameter in determining testing complexity.* Hence we manipulate equation 2 yielding:

$$(1 - \beta_x)^{n_x} = 1 - \alpha$$

$$n_x \ln(1 - \beta_x) = \ln(1 - \alpha)$$

Since we want a discrete n_x, we solve for n_x and apply the ceiling function (equation 4).

$$n_x = \left\lceil \frac{\ln(1 - \alpha)}{\ln(1 - \beta_x)} \right\rceil \tag{4}$$

Assuming our latent failure rate estimate is non-zero and positive, we select n_x input points from the same distribution used during sensitivity analysis from the input partition for path equivalence class x. So n_x is the number of input points used to show that location x is "approximately" correct with confidence α in our assertion. A non-positive latent failure rate estimate suggests testing may not be the best scheme to gain confidence in the code; another technique may be appropriate.

Our testing complexity metric, Υ, describes the complexity involved in testing a program by producing the number of points required to achieve a certain confidence in the assertion that the program is "approximately" correct. We define Υ to be:

$$\Upsilon = \sum_{z \in \{PEC\}} n_z. \tag{5}$$

where PEC is the set of all path equivalence classes of a program.

To see how equation 5 performs, consider a program P with four path equivalence classes i, j, k, and l, where $\beta_i = 0.01$, $\beta_j = 0.02$, $\beta_k = 0.009$, $\beta_l = 0.5$, and $\alpha = 0.999$. Then $\Upsilon \approx 1.8\text{x}10^3$. Now consider a program K with one path equivalence class, and a latent failure rate estimate of 0.00001. Then $\Upsilon \approx 6.9\text{x}10^5$. Although P has more path equivalence classes and is possibly more structurally complex, it requires fewer test points because it tends not to hide faults. Program K, however, tends to hide hides faults, and therefore is harder to test.

4 Guiding Testing

Section 3 described the degree of testing necessary to assert that a program is "approximately" correct with a particular confidence. This section again uses the notation defined in Appendix A to explain how to determine the degree of testing a particular location needs relative to other locations. This information is useful when testing resources are limited. If we jump back to our mining analogy with a new goal of trying to satisfy the original goal at as many locations as possible instead of for the entire mine, the information provided in this section is equivalent to the geologist saying, "Since you only have so much strength, don't dig at that location, because the gold will be very deep if it exists and you will tire yourself. You can easily dig at these locations instead."

For two distinct program locations x and y, if $\beta_x \ll \beta_y$, then location x should receive either more testing than location y or possibly *specialized* testing.[4] As an example, consider the following two locations in a program: i := i + 1 and ((slope*x1 + x2/slope - y1 +y2)* slope)/(slope**2 +1)). The first location has a zero latent failure rate estimate assuming independence, and a negative latent failure rate estimate assuming non-independence (under a particular input distribution). In this case, the geologist might say "abandon hope for this area– it's about to cave in!" The second location has a latent failure rate estimate of 0.0416 assuming independence and a latent failure rate estimate of 0.0413 assuming non-independence (also under a particular input distribution). From these estimates, the first location should receive more testing since it has shown the potential for never revealing a fault even though a fault may exist at that location. In fact, this location should be rewritten or analyzed other than by testing. The second location has shown the ability to reveal a fault, hence it is a less worrisome location for testers.

[4] [7] contains a scheme for isolating input points that have previously exhibited more tendency to reveal failures. Using such input points is what is meant by *specialized* testing.

5 Summary

The testing complexity metric produced by equation 5 refutes the notion that more locations necessarily means greater testing complexity. This metric instead considers testing complexity to be relative to the minimum failure probability estimate over the set of all path equivalence classes of the program. For non-positive latent failure rate estimates, the corresponding path equivalence classes must be ignored, and equation 5 must exclude summing over these path equivalence classes. This situation causes an "incomplete" complexity metric; investigation is continuing into other testing complexity metric models for path equivalence classes having non-positive latent failure rate estimates that can be incorporated in equation 5. One solution to non-positive latent failure rates is to use the formulae in Section A.1 assuring a zero or greater latent failure rate estimate.[5] Experiments using sensitivity analysis for debugging [3, 7] have shown that in general the independence assumption used in the formulae of Section A.1 holds. In fact, it may be extremely rare for the non-independence phenomenon to occur.

This paper has shown that sensitivity analysis may be used as a new structural testing scheme, both in quantifying how much testing to require, as well as where to perform testing when resources are limited. This structural testing scheme requires no oracle or specification, meaning that it can decide where to concentrate testing and to what degree without checking the output. However once testing begins, it is necessary to have a way of checking the output.

6 Acknowledgements

The authors would like to thank Keith Miller for reviewing earlier versions of this document.

References

[1] Edward Miller and William E. Howden. *Tutorial: Software Testing and Validation Techniques.* IEEE Computer Society Press, 1981.

[2] Jeffrey M. Voas and Larry J. Morell. Fault sensitivity analysis(PIA) applied to computer programs. Technical Report WM-89-4, College of William and Mary in Virginia, Department of Computer Science, December 1989.

[3] Jeffrey M. Voas and Larry J. Morell. Propagation and infection analysis(PIA) applied to debugging. *IEEE Southeastcon '90 Record*, April 1990.

[4] Stephen R. Schach. *Software Engineering.* Aksen Associates Incorporated Publishers, 1990.

[5] See Appendix A.

[5] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. *Second Workshop on Software Testing, Validation, and Analysis*, pages pp. 206–215, July 1988.

[6] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):pp.1134–1142, November 1984.

[7] Jeffrey M. Voas. *A Dynamic Failure Model for Performing Propagation and Infection Analysis on Computer Programs.* PhD thesis, College of William and Mary in Virginia, March 1990.

A Latent Failure Rate Estimation

Appendix A contains formulae for estimating the minimum failure probability for a single location, a path equivalence class, or a program. A *path equivalence class* is defined to be a grouping of paths that have some similar property. In our work, this property has been having the same locations reached. The set of all path equivalence classes of a program represent all paths through the program for a particular set of input points, hence our set of path equivalence classes is an *equivalence relation* on the set of program paths.

This appendix assumes *prior* knowledge of how to perform propagation and infection analysis; the results of propagation and infection analysis are the key indicators in determining sensitivity. This information is available in [2, 7, 3]. Before we begin, we should state one important assumption: *the propagation estimates and infection estimates of a particular location have been the function of an input set that is equally representative of the path equivalence classes on which the location is a member.* Without this assumption, it becomes necessary to define the propagation and infection estimates for a location as a function of a particular path equivalence class. Thus if location l occurs on three path equivalence classes, then there must be three estimates for location l, one for each path equivalence class.

The notation used in the formulae of this appendix follows:

$\min_k[I_{l,a_k}]$ represents the minimum infection estimate over the entire set of semantic alternatives $(a_1, a_2, \ldots, a_n)$ used at location l,

$\min_z[\mathrm{F}_{l,a_z}]$ represents the minimum propagation estimate over the entire set of active variables $(a_1, a_2, \ldots, a_n)$ upon which perturbation functions are applied at location l, and

β_l represents the latent failure rate estimate for either a single location, path equivalence class, or program, depending upon the interpretation of l.

A.1 Latent Failure Rate Estimation Assuming Independence

The first attempts at estimating the minimum failure probability are found in equations 6- 8. Equations 6- 8 assume independence between input points. This means that we assume that

an input point infects independent of whether it propagates, and an input point propagates independent of whether it infects. It is not certain whether this assumption always holds; the independence property is in some sense a function of how the estimates are determined. If the infections which occur during infection analysis are the infections used during during propagation analysis, then the independence property for these estimates will hold [7]. However since a perturbation function [7, 2, 3] is applied in our algorithm for determining propagation estimates to avoid the criticism levied against fault seeding techniques, the property can not be arbitrarily assumed to hold for all programs.

So the latent failure rate estimate for a location l assuming independence is:

$$\beta_l = \min_k[I_{l,a_k}] \cdot \min_z[\mathrm{F}_{l,a_z}] \tag{6}$$

Equation 6 can be generalized to a path equivalence class l with:

$$\beta_l = \min_{i,j}[\min_k[I_{j,a_k}] \cdot \min_z[\mathrm{F}_{i,a_z}]] \tag{7}$$

where locations i and j are reached on each path represented by path equivalence class l. Notice that equation 7 does not require location j to be a successor location of location i. Since it is intuitive to think of infection occurring before propagation, equation 8 takes this into account producing a less conservative estimate:

$$\beta_l = \min_{i,j}[\min_k[I_{j,a_k}] \cdot \min_z[\mathrm{F}_{i,a_z}]] \text{ where } i \text{ precedes } j \tag{8}$$

In equation 8, predecessor/successor relationships between locations are determined dynamically.

A.2 Latent Failure Rate Estimation Assuming Non-Independence

Section A.2 describes a more conservative approach to estimating the minimum failure probability than is produced by equations 6 - 8. It does not assume that those points which infect will propagate. It considers a class of input points which are referred to as non-propagators. A *non-propagator* is an input point which will cause an infection at a location but will not propagate from that location. Equations 9 - 11 subtract the proportion of non-propagators from the infection estimate, leaving an estimate of the proportion of the input points that will infect and propagate. This non-independence approach is more conservative than the independence approach; the testing complexity metric relies on having the most conservative estimate of the minimum failure probability, however being too conservative and arbitrarily assuming non-independence can lead to unusable latent failure rate estimates in the testing complexity metric.[6] Hence Section A.1 can not be arbitrarily dismissed as unrealistic, although its estimates are more liberal in some cases. For a testing complexity metric and

[6] A latent failure rate estimate is termed ***unusable*** if it is zero or negative because it can not be used for the testing complexity metric.

other software analysis applications such as software reliability, having the most conservative estimate is crucial.

The latent failure rate estimate for a location l assuming non-independence is estimated by:

$$\beta_l = \min_k[I_{l,a_k}] - (1 - \min_z[\mathrm{F}_{l,a_z}]) \tag{9}$$

Equation 9 effectively removes the proportion of input points which for at least one perturbed active variable did not propagate. Equation 9 can be generalized to a path equivalence class. The "most conservative" estimate of the latent failure rate for a path equivalence class l is given by:

$$\beta_l = \min_{i,j}[\min_k[I_{j,a_k}] - (1 - \min_z[\mathrm{F}_{i,a_z}])] \tag{10}$$

where locations i and j are reached on each path represented in path equivalence class l. If the same requirement from equation 8 is made for equation 10, the latent failure rate estimate becomes:

$$\beta_l = \min_{i,j}[\min_k[I_{j,a_k}] - (1 - \min_z[\mathrm{F}_{i,a_z}])] \text{ where } i \text{ precedes } j \tag{11}$$

Again in equation 11, predecessor/successor relationships between locations are determined dynamically. Equation 10 uses the lowest propagation estimate for any location i in path equivalence class x and the lowest infection estimate for any location j in l. Equation 11 does also provided that location i precedes location j.

Both equations 9, 10, and 11 can produce negative latent failure rate estimates. This is due to the subtraction of the percentage of "nonpropagators." Although a latent failure rate estimate which is less than or equal to zero is useless as a probability, such an estimate contains useful information. A negative or zero latent failure rate estimate says that the location may potentially (worst case) always produce the correct output even when there is a fault in it. Thus this location may hide a fault that can never be found during testing.

A.3 Latent Failure Rate Estimation for an Entire Program

The latent failure rate estimate may be determined for an entire program. Our latent failure rate estimate for a program P is:

$$\beta_P = \min_{i \in PEC}[\beta_i] \tag{12}$$

where PEC represents the set of path equivalence classes of program P. Equation 12 is the minimum latent failure rate estimate over all path equivalence classes; The "conservativeness" or lack thereof of equation 12 depends on the estimation scheme of the latent failure rate chosen at the path equivalence class levels. A more conservative approach to equation 12 is:

$$\beta_P = \min_{i,j}[\min_k[I_{j,a_k}] - (1 - \min_z[\mathrm{F}_{i,a_z}])] \tag{13}$$

where locations i and j are in program P.

Scenario Oriented Engineering for Software Intensive Systems

Michael S. Deutsch

Hughes Aircraft
Building 618 MS B204
P.O. Box 3310
Fullerton, CA 92634

Abstract

Scenario-oriented engineering is an approach to full-lifecycle software engineering. A "scenario" defines a continuous path from an external stimulus into a system through a response back into the external environment, a behavior pattern that would be visible to a system user, and is a powerful macro-object that can be used to conceive, define, design, validate, test, and maintain a system. This presentation is structured by project success factors from an empirical study of 25 projects.

Biographical Sketch

Michael S. Deutsch is a Chief Scientist in Hughes Aircraft Company's Ground Systems Group, where he is responsible for long range software engineering technology planning. He is best known for his work in software quality methods and is the author of two textbooks in this area. His present work focuses on empirical studies of the software project management process. He serves on the editorial board of IEEE Transactions on Software Engineering.

Scenario Oriented Engineering for Software Intensive Systems

MICHAEL S. DEUTSCH
Chief Scientist
HUGHES AIRCRAFT COMPANY
Ground Systems Group
Fullerton, California

HUGHES

port

What Is A Scenario?

HUGHES

- Depicts user perceivable behavior from stimulus to response
- Use as a common system view from concepts through maintenance
- Enhances communications between user, developer, and customer
- Supports evolutionary life cycle processes

port

Discussion Topics

HUGHES

Properties of scenarios & examples

Multi-layer scenario architecture model

Process of scenario oriented engineering

Empirical effectiveness study

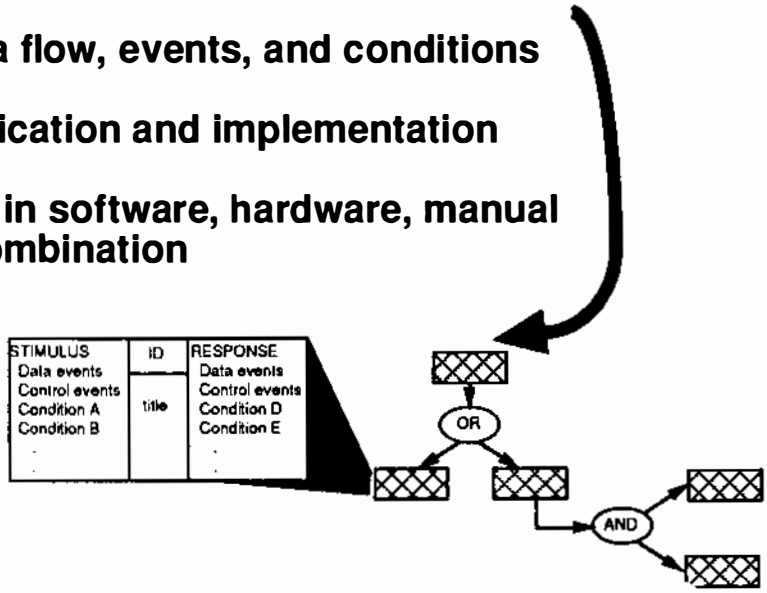
Air traffic control system architecture

port

Properties of Scenarios

HUGHES

- **Depicts user perceivable behavior path from from stimulus to response**
- **Contains data flow, events, and conditions**
- **Both a specification and implementation**
- **Implemented in software, hardware, manual actions, or combination**
- **Is *informal,* but .....**



port

Air Traffic Control Scenarios

HUGHES

To be supplied

port

Multi-Layer Scenario Architecture Model

HUGHES

To be supplied

port

Scenario Engineering Process

HUGHES

- Fuzzy conceptual thinking documented in representative scenarios
- Mental rehearsal reveals off-nominal, failure, or anomalous mutations
- Risky/uncertain scenarios candidates for prototyping and/or in-depth analysis
- Subset of scenarios is first increment of operational system
- Probable domain-specific architectures
- User, customer is full partner in process

port

Unprecedented System Characteristics

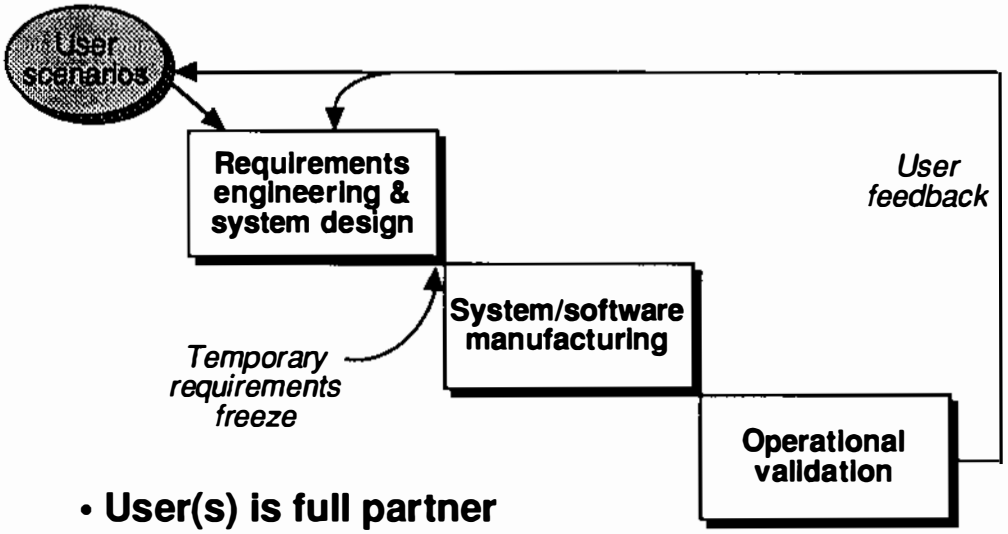
HUGHES

- Specifics of doctrines, threats, scenarios, and constraints subject to frequent change
- Operational requirements, acceptance criteria, and other measures cannot adequately be specified in advance
- Numerous complex and changing interfaces

port

Evolutionary Process Framework

HUGHES



- User(s) is full partner
- Isolation of research from development
- Incremental incorporation of new technology

port

System Architecture
Air Traffic Control System

HUGHES

To be supplied

port

Exploratory Data Analysis

HUGHES

Goal: examine feasibility of conceptual model

Data on 25 projects collected using informal questionnaire

Caveats on results

- **Insights are pointers for future study**
- **No statistical inferences**

port

Scenario Effectiveness Variables

HUGHES

Subset of 55 success factors in correlational study:

- How completely were user operational scenarios defined?
- To what degree were requirements and system design validated the user operational scenarios?
- To what degree were there ongoing collaborative contacts between users, customer, and developer to assure correct content in the technical requirements?
- To what degree were user operational scenarios included in system and acceptance testing?

port

Intercorrelations: Scenario Factors With Project Performance

HUGHES

Factor	All projects		Adverse projects	
	technical performance	*business performance*	*technical performance*	*business performance*
Degree of scenario specification	0.62	0.73	0.52	0.77
Degree of dialogue	0.68	0.66	0.75	0.61
Degree of reqmts/design validation	0.55	0.59	0.70	0.77
Degree of scenario testing	0.37	0.41	0.61	0.64
Combined four factor score	0.70	0.75	0.77	0.81

port

Key Points Summary

HUGHES

- Scenario orientation across lifecycle enables effective communications with system stakeholders
- Prospect for inclusion of user scenario view in system architecture
- Recent software technology advances support scenario/architecture correspondence
- Some evidence of empirical effectiveness of scenario orientation
- More work needed in domain specific architectures and distributed communications

port

FEATURE POINTS GUIDE REAL TIME SOFTWARE DEVELOPMENT

James K. Larson

Senior Hardware/Software Engineer
Television Division, Tektronix, Inc.
PO Box 500
Beaverton, OR 97077
503/690-7037 Email: jamesk@gemini.TV.TEK.COM

ABSTRACT

Effective management of any software project requires close attention to specification, quality, and schedule. Of these, the one most likely to adversely impact the other two is schedule. Specifications are often revised and quality is commonly sacrificed because time runs out on a project. Significant features often are deleted because their impact on the schedule is uncertain. Schedule is also very difficult to control and predict. In this paper the concept known as feature points is shown to be an effective tool to predict schedule for a real time embedded system. The tool can be used to predict both the completion time and the impact of product feature set changes. It is constantly updated by the most recent project performance data and thus improves its prediction accuracy as the project progresses. Since this results in accurate and reliable scheduling estimates, management and project trust builds, product changes have known impacts on schedule, and the time required to do a quality job and meet all specifications is known. Improvement of quality results since surprise avoidance and better schedule projection allow more orderly development.

In this paper the concept of feature points is extended to the design and tracking of real time systems. In contrast to the traditional feature point method, the method described here derives a project-specific metric, then uses it to schedule and track that project. Whether the feature point per man-month values can be extended to other projects or project teams is not known. An example and case study are presented to show the application and results on an actual project. This paper describes the definition and use of feature points to guide software development and scheduling.

Keywords: Estimating Schedule for Real Time Software, Project Management, Real Time Software Development, Feature Points, Software Quality.

About the Author

James K. Larson received the B.S. in Mechanical Engineering from New Mexico State University in 1972 and the M.S. in Biomedical Engineering from the University of California at San Diego in 1984. He has been involved with real time embedded systems since 1980. He is currently a Senior Hardware/ Software Engineer with Tektronix where he develops television waveform test and measurement instruments.

Introduction

Feature points[1] are a measure of the work content of software. They are independent of the implementation language and the number of lines of code. Features are defined as significant components of the software. The functional specification and a structured analysis diagram are used to define the modules and then to estimate the feature points for each such module. Definition and estimating are most consistent if done by one person using criteria that are as consistently applied as possible. The six feature types used are algorithms, inputs, outputs, inquiries, data files, and interfaces. These are defined for real time, embedded systems with a point weighting for each feature type assigned. Examples of using system structure charts and functional specifications to derive feature points for a system are given. Feature points may be effectively used to evaluate the state of software development, the rate of progress and the projected completion date. Using them, it is possible to include such convolved factors as hardware interaction, real-time requirements, etc.

The feature point method encourages breaking the project into manageable and measurable steps, shown to be an effective approach to systems design[2], especially for systems in which all requirements are not known at the project start. As a skeletal or prototype system is developed the feature points and man-months are tracked. A prototype effort then yields a prototype scheduling tool as well as a prototype product to guide future development work. The feature point method works well with the development method (incremental improvement) of Basili and Turner. Using the initial "small, but functional" system, a feature point/ man-month metric may be developed. This metric may be used to estimate and track the remaining project. As changes are required, their impact on the schedule may be estimated. A manpower curve for the project may also be developed if a desired schedule is established. The effect of adding functionality to the system can also be quickly assessed.

Using a case study, the feature points for a prototype system were calculated and a prototype estimate in feature points per man-month was calculated. This represented a 1 year time frame and 18 man-months. Using this prototype estimating tool and the full system functional specification and structure diagram, completion time for the project was estimated. (Since the actual manpower curve is now known, an actual completion time is known and the error can be found.) At this stage in the case study, based only on the software productivity during prototype development, the error in estimated completion time was about 4 months (high) in an 18 month development time. This time estimate established the feasibility of the project but also showed that more manpower was required. Further tracking through the rest of the project, with reassessments at 3 to 4 week intervals, showed that with 3 months to go, the error in the estimate was essentially zero. Error in the prototype estimate was probably due to the fact that 1 year of data was collected and used all at once instead of using more frequent measurements.

Currently, feature point estimates are validated only for estimates made by one person and applied to one project. Whether they can be extended beyond this is unknown, but even with these restrictions they are a very useful tool for scheduling and can have a significant impact on product quality.

---

1 *A Short History of Function Points and Feature Points*, Capers Jones, McGraw Hill, 1986.

2 "Iterative Enhancement: A Practical Technique for Software Development", V. Basili and A. Turner, IEEE Transactions on Software Engineering, Dec., 1975

Feature Definitions:

Algorithm

An algorithm is any calculation procedure. For example, the method of calculating a digital to analog converter setting from a multi-turn potentiometer reading is an algorithm. So is the method of mapping a potentiometer input to the correct digital to analog converter output in a multiplexed input scheme. Each algorithm is given 6 feature points.

Input

An input is considered a message passed from another task or an interrupt service routine. For example, a message from the front panel interrupt handler to the front panel interpreting task that the front panel state has changed is an input. Each input is given 4 feature points.

Output

An output is defined as a message passed to another task. For example, a message to the task that updates the display that a front panel change has occurred and that the new state is decoded and available is an output. Each output is given 5 feature points.

Inquiry

An inquiry is defined as an access to a significant data structure. For example, to update the instrument display after a front panel change requires an inquiry into the data structure holding the instrument state. Each inquiry is given 4 feature points.

Data File Creation

A data file creation is defined as creation of a significant data structure. For example, updating the instrument state data structure after a front panel change is a data file creation. Each data file creation is given 4 feature points.

Interface

An interface is defined as an access to an external hardware component. For example, reading the raw front panel button state is an interface. So is configuring the latches that control data acquisition. Each interface is given 7 feature points.

Summary

Features and their associated points are summarized in the following table.

Feature	Points
Algorithm	6
Input	4
Output	5
Inquiry	4
Data File	4
Interface	7

Complexity Adjustment

It is possible to use a complexity adjustment (suggested range: 0.6 to 1.4) to reflect the fact that not all features are created equal. This could be useful to weight hardware interactions or complex algorithms more heavily. For the example in this paper, no complexity adjustment was made. That is, a weight of 1 was given to all features.

Case Study

System Definition and Team Description

The system for which this feature point analysis was done was an electronic test instrument which could acquire and display data in real time. Total development time was about two years. From 1 to 3 software engineers worked on the project. Figure 1 shows the actual manpower curve.

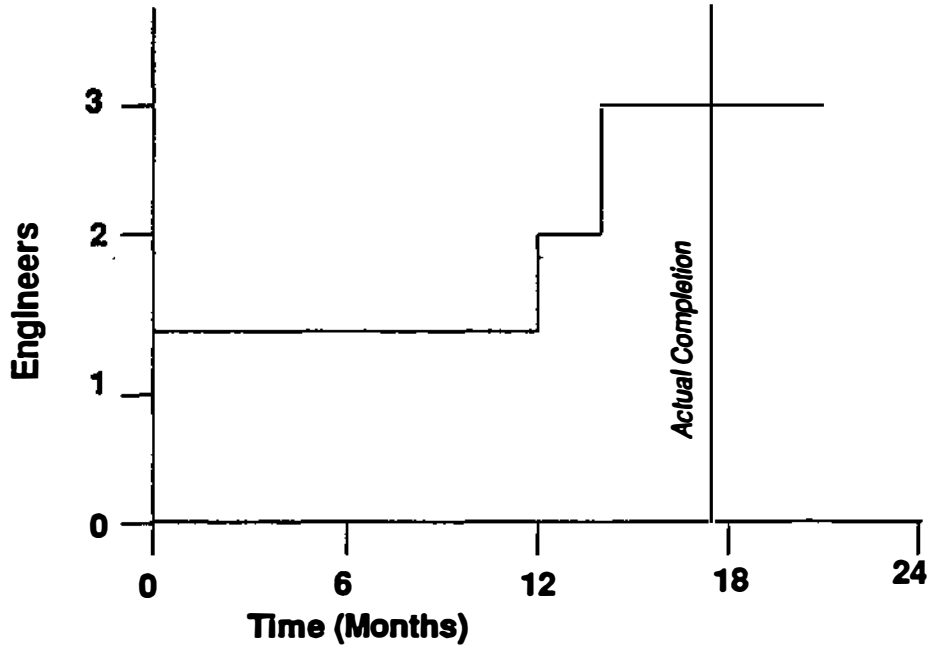


Figure 1

Actual Manpower Curve

The feature point method in this paper was developed in response to a need to determine the feasibility of the project schedule, to determine software manpower requirements and to track the development effort.

The system software consisted of about 25,000 lines of C code. Personnel ranged from experienced software engineers (7 plus years of experience) to a new college grad having only co-op experience.

The instrument had to respond in real time, collecting and displaying data and handling input from the front panel and the communication port. The specifications for the software consisted of a detailed functional specification and a high level structured analysis chart. Figure 2 shows a portion of the structure chart indicating the level of detail in the actual document.

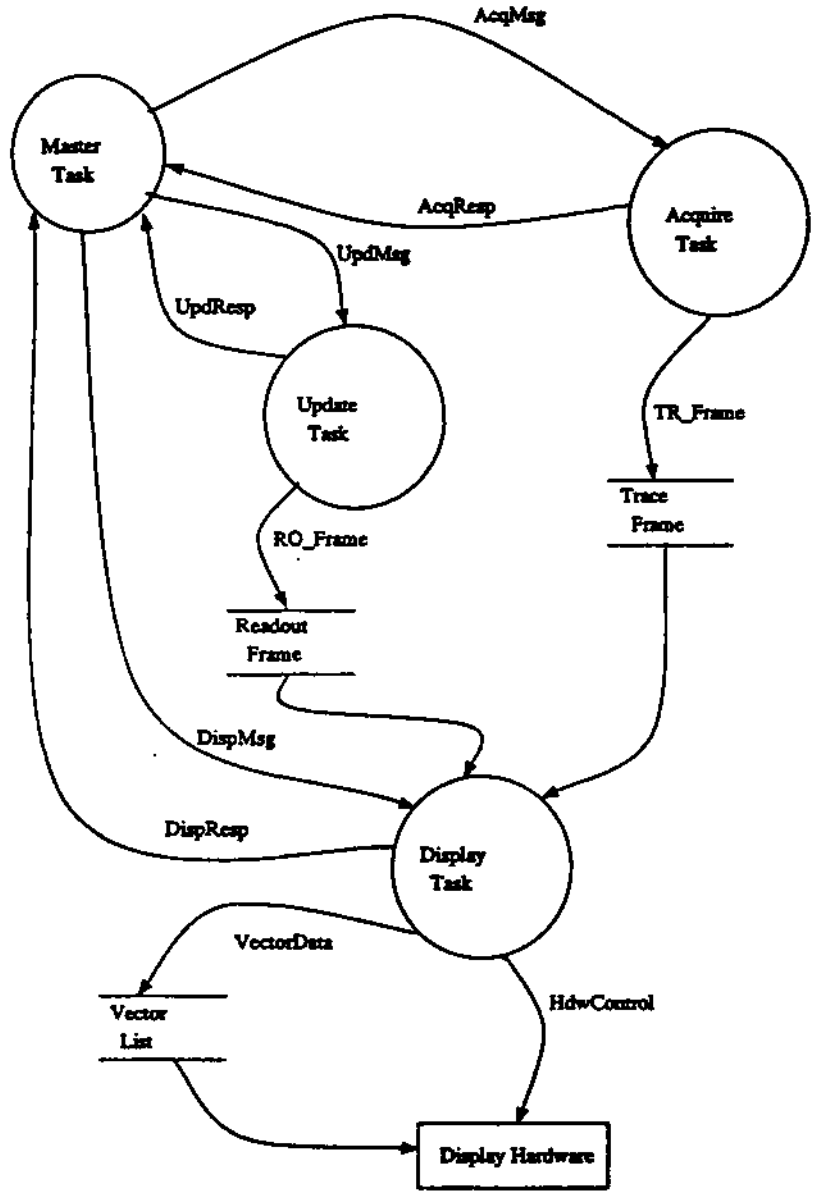


Figure 2

After a one year development effort, a partial prototype existed which had most major functions (hardware and software) partially working. At this point, the feature point system described herein was devised and applied to assess the possibility of completing the project on time and to determine what additional manpower might be required.

Examples of Feature Point Estimates

Estimate from Functional Specification

As an example of feature point estimating, consider the feature points resulting from the implementation of the following excerpt from the functional specification. "The current selection shall be indicated by a box drawn around it in the readout display." In order to draw a box around an item, it is first necessary to check the front panel structure (an inquiry into a data file) to know where to put the box. Then, an algorithm must exist to draw the box as a series of vectors.

Finally, the vectors must be put into the readout vector list (a data file creation). Total feature points: 6 + 4 + 4 = 14.

Estimate from Structure Chart

As a further example of feature point estimating, consider the feature points resulting from the implementation of the Display Task shown in the partial system structure chart of Figure 2. Two algorithms are required, one to convert the readout characters into vectors and one to convert the vectors into data for the display controller hardware. There is one input from the Master Task (the output is a simple response - not counted). Two inquiries of data files are required and one data file is created. One interface to hardware exists. Total feature points: (2x6) + 4 + (2x4) + 4 + 7 = 35.

Results

Feature points for a 1 year prototype development were compiled by defining the tasks and functions completed and assigning feature points to them. The remaining tasks and functions were also defined and feature points computed for them. Using the feature points per man-month for the prototype and the feature points remaining, the schedule could be projected. As a result, the decisions to shift the priorities of one engineer to full-time software development and to hire another software engineer were made. *

Tracking of feature points and man-months was handled simply by a yellow tag system. A chart showing each engineer's name with space to post the tasks assigned to him was maintained. Tasks were written on yellow tags ("Post-Its") along with the feature points for the task. These were placed next to the name of the engineer assigned to the task. As an engineer started a project he noted the date. The date of completion was recorded when the task was functional. The results were compiled every few weeks and a report was generated. One such report is shown as figure 3. The feature point per man-month rate was calculated for the project to date and the new completion date was projected. In this manner the project progress was tracked and the feasibility of timely completion could be constantly reassessed.

Schedule Summary

Rating the Instrument Software on a feature point scale: (17-Jun-88)

- ✓ 205 Feature points completed by three engineers in 7 weeks
- ✓ 121 Feature points to go
- O Yearly average rate is 35 feature points / person month
- ✓ At present rate using 3 Engineers:
  - ☛ 1 month to complete software!

Figure 3
Feature Point Tracking Report

The actual manpower curve for the project is shown in figure 1. This actual curve was used to make table 1 which shows the improvement of the feature point per man-month estimate and projected completion versus actual completion for the project. With 3 months to go, the estimated completion is essentially identical to the actual completion. This level of accuracy might have

* A modification to the functional spec well into the project added feature points. This complication is not explicitly shown here for the sake of simplicity and clarity. The feature point method both estimated the impact of the change and adjusted smoothly to the added complexity.

occurred earlier if the tracking of the prototype development(first year) had been more accurate.

The table below summarizes the results of tracking the project. Note especially the error between actual completion date and predicted completion date.

Table 1

Elapsed Time (Months)	Completed (Incremental Featr Pts)	To Go (Total Featr Pts)	Avg Rate (FPts/MM)	Completion Time (Months)	Actual Compl Time (Months)	Error (Months)
0	1141	1141	-	-	-	-
12	465	612	26	9.5	5.5	4
12.5	78	534	29	7.5	4.5	2.5
13.5	130	456	33	5	4	1
15	142	326	35	2.5	2.5	0
16.5	205	121	35	1	1	0
17.5	121	0	36	Done!	Done!	0

Analysis

After collecting data for 12 months during prototyping, the feature point method was used to estimate project completion. The estimated completion times at various staffing levels shown in Figure 4 established both the feasibility of completing the project on schedule and the urgency of hiring additional manpower to help insure timely completion.

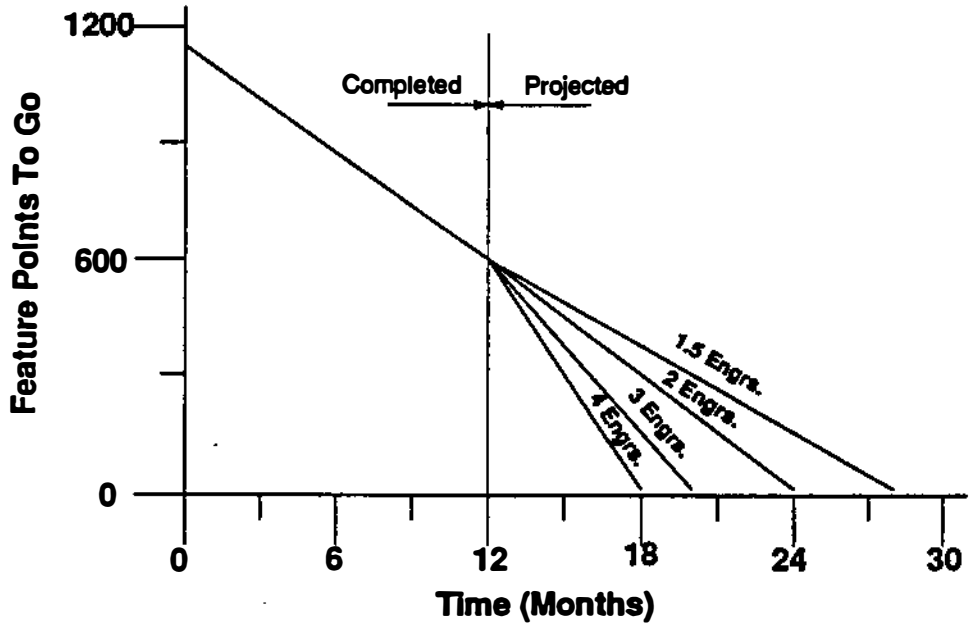


Figure 4

Projected Completion Time

The tracking reports shown in Figure 3 kept management informed of progress and confident that schedule estimates were reliable.

The impact of specification changes was readily estimated and put decisions about adding or removing features into perspective.

In conclusion, the feature point system described in this case study proved to be an effective scheduling and tracking system. It was easy to apply and the results were more than acceptable.

Restrictions of this Model:

The main weakness of the feature point system application in this paper is that it was not applied to the system debug phase. There were several reasons for this; none of them were related to deficiencies in the feature point system, but rather to personnel changes and a severe time crunch.

One obvious restriction of this case study is that the product software team was only three engineers. How this method will work for larger teams is unknown. It may be necessary to weight the feature points for a given module based on which engineer is implementing it. It is important to do this consistently.

The feature point system used here also assumes that feature points per man- month during prototyping are equal to feature points per man-month during final development. This assumption may not be fully justified - especially if the team is changed. The feature point per man-month metric is continually updated as a moving average throughout the project and thus any strong variations in productivity will eventually be reflected as a change in the metric. Extrapolation to other projects is risky and should be done cautiously.

Use of Feature Points for Project Management

Applications in Project Management

Three important aspects of project management must be considered: software quality, scheduling and specification.

Software Quality

Since quality so often suffers because time runs out or because of constant time pressure (there's never time to do it right, but always time to do it twice), the importance of accurately and reliably predicting schedule is obvious. The feature point approach in this paper is a method to do just that. Knowing the impact on the schedule of rewriting or changing the software greatly improves the likelyhood of having time granted to do things right.

The feature point system also emphasizes and reinforces team effort. Since the productivity of the team is clearly visible, each team member takes more interest and pride in team accomplishment. The confidence and trust of the rest of the project team and project management grows as the software team is able to accurately schedule their work.

Scheduling

As explained in the case study, the feature point method helps to estimate schedule and completion time, it also serves to track the development effort. Further, it can be used to predict manpower requirements. Finally, it encourages breaking the development effort into many small projects (chunks) which are more easily completed and tracked.

Specification

The feature point method allows evaluation of the feasibility of meeting a desired schedule after a specification is written, but far in advance of completion of the project (which is when schedule is often known today). Proposed specification changes or additions can now have an accurate schedule impact known at the time they are proposed.

Recommended Practice

Plans for effective use of the feature point technique on future projects involve combining it with the method of Basili and Turner (op. cit.). The recommended procedure is described below.

Step 1: Do a simple implementation of a skeletal subproblem. That is, build a prototype. The prototype should contain a good sampling of the key aspects of the final system, allow major hardware elements to be exercised, and demonstrate solutions to major known problems. It should be simple enough to understand and implement easily. The result should be usable and useful to a potential user. The time spent writing this software must be carefully recorded.

Step 2: Create a project control list showing all software modules which must be created to produce the final product. This is most easily done after a complete functional specification is written and a high-level structured analysis diagram is drawn. The functional specification should fully describe the interface to the user and to any system hardware. The structured analysis should be sufficiently detailed to show the major software tasks and their data flows. The modules required to complete the project can now be defined and the project control list compiled. As a rule of thumb, a maximum sized module would require about 3 weeks work to design, test and debug. Feature points should now be assigned to each module. This can be done by one person, or averaged from results done by team members. It is important that feature point definitions be clear and any weighting be done consistently.

Step 3: Identify the modules in the prototype and assign feature points using criteria consistent with that used in Step 2. The feature point per man-month metric for the prototype effort can now be computed. Using this metric and the feature points compiled for the project control list, an initial estimate of the project schedule can be made. At this stage, it is important to keep in mind that this development method is iterative - that is, some modules may require rewrite which will add feature points to the project. What allowance should be made for this can only be established by experience. An initial estimate of 25% is proposed.

Step 4: An analysis of each new module proceeds according to the criteria of Basili and Turner. Any additions to the project control list will have an estimated schedule impact which can be measured by the latest feature point per man- month metric. Use a yellow tag or similar system to keep careful records and update the feature point per man-month metric as a moving average about every 3 weeks.

Step 5: As the system is integrated and system level bugs appear, **keep using the feature point system.** For any module that must be rewritten, use the feature points for that module to estimate the schedule impact. If module interactions are buggy, then use the total feature points for all interacting modules to estimate the time to fix the problem.

Possible refinements could include the use of weighting factors, but this must be done in a consistent manner. Another possibility is that future projects may be able to apply the feature point per man-month metrics from existing projects, especially if they are essentially the same product and the team is essentially the same.

Conclusion

The feature point system as described in this paper is shown to be an effective tool for scheduling and tracking a project. Starting with prototype development, an estimate of feature points per man-month is found. Using the functional specification and a structure diagram, the project schedule is projected. This schedule estimate is specific to the project and is independent of implementation language. Further tracking of the project using feature points shows the progress of the project and refines the accuracy of the feature point per man-month metric for the project. The impact on the schedule of any changes proposed or required can be estimated quickly. Having an accurate estimate of schedule has many beneficial effects on the quality of the resulting product.

SOFTWARE INSPECTION, VERIFICATION, AND TESTING A HYBRID APPROACH

Daniel Hoffman
University of Victoria
Department of Computer Science
P.O. Box 1700
Victoria, B.C., Canada V8W 2Y2
(604) 721-7222
dhoffman@uvunix.uvic.ca

KEYWORDS

software quality, software inspections, program verification, software testing, formal methods

ABSTRACT

This paper presents a novel approach to improving software quality using a combination of existing techniques. We review the underlying techniques, describe the combination scheme and summarize our experience in applying the scheme. The key contribution is a set of pragmatic criteria for software specification and proof, and a demonstration that these criteria are effective in practice.

BIOGRAPHICAL SKETCH

Dr. Daniel Hoffman received the B.A. degree in mathematics from the State University of New York, Binghamton, in 1974, and the M.S. and Ph.D. degrees in computer science in 1981 and 1984, from the University of North Carolina, Chapel Hill. From 1974 to 1979 he worked as a commercial programmer/analyst. He is currently an Assistant Professor of Computer Science at the University of Victoria. His research area is software engineering, emphasizing software specification and testing.

SOFTWARE INSPECTION, VERIFICATION, AND TESTING A HYBRID APPROACH

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

1 UNDERLYING TECHNIQUES

1.1 Specification and Code Inspections

According to Fagan [1], *software inspection* is an effective and economical method for improving software quality. Fagan's approach requires that the software development process be broken into a sequence of steps, with explicitly and precisely defined criteria for the product of each step. In inspection meetings, the participants examine the product in detail, attempting to find *faults*: deviations from the criteria. We distinguish, as Fagan does, between inspections and *walkthroughs*. Inspections depend on precise criteria and focus on identifying faults. Walkthroughs typically do not require precise criteria and the goals are broader, including identifying faults, and discussing design alternatives and coding practices. For our purposes, the narrow focus of inspections is preferred.

Fagan's arguments and experimental evidence are persuasive. Yet, he has relatively little to say about the software development steps and criteria to be used.

1.2 Defining the Software Development Phases

Parnas presents a sequence of software development phases, based on sound principles and demonstrated on substantial examples [2]. The key themes are the use of complete, formal specifications of observable behavior [3], and the reduction of maintenance costs through *information hiding* [4]. While the Parnas method is appealing, there are two impediments to its widespread use: (1) the use of completely formal specifications, as currently understood, is impractical and (2) the method provides little support for showing that implemented code meets its specification.

1.3 Program Verification

The work of Floyd [5], Dijkstra [6] and many others has provided a fundamental understanding of programming and the programming process, and methods for performing formal program verification. Furthermore, the argument that correctness can only be demonstrated through proofs is convincing, given *Dijkstra's Law of Testing*: "Program testing can be used to show the presence of bugs, but never their absence [7]."

Despite its maturity, program verification research has had disturbingly little impact on industrial practice. This is due in part to two weaknesses in the Computer Science view

* Research supported by the Natural Sciences and Engineering Research Council of Canada under grant A8067.

of formal verification, discussed in detail in the controversial paper by DeMillo, et al. [8], and also by Naur [9]. The first weakness is that completely formal program proofs do not *guarantee* correctness and never will. For non-trivial programs, such proofs are extremely complex and are thus error-prone themselves. The second weakness lies in the claim that formal program verification will do for program correctness what mathematical proofs do for theorems. The problem is that mathematicians rarely resort to formal proofs, for good reason. According to Karp, "Once a mathematical truth is discovered, the 'proofs' used to communicate it from one human to another are never formalized completely, if only because such formalization is a hideously tedious process and because the resulting formal proofs would be opaque to human readers [10, pg. 1411]." So, instead of relying on formal proofs, mathematicians use *informal* proofs, and wide and repeated review by other mathematicians. It is the social process as much as the proof on which they base their confidence in the truth of the theorems.

1.4 The Impact on Industry

The result of the insistence on formal specification and formal program verification is a crippling stand-off between the "formalists" and software developers, academic *and* industrial. In practice, even informal methods of specification and verification are rarely used, leaving software developers to struggle to control complex systems without the most basic mathematical support.

2 A HYBRID APPROACH

2.1 The Conceptual Basis

We have developed an approach that combines the above techniques in a practical way.

1. We work primarily within the framework developed by Parnas. For each of the *work products* proposed by Parnas (plus a few more), we have developed work product criteria, describing the purpose and evaluation criteria for the product.
2. We use Fagan inspections, reviewing every work product in detail, to identify faults and reinforce the understanding of the criteria by the inspection teams.
3. We use selected verification techniques to demonstrate the conformance of implementations to specifications.

In the previous section, we described the difficulties involved in applying formal specification and verification to complex software systems. We have addressed these problems through pragmatic criteria for specifications and proofs. We make the following claim, the central hypothesis of this paper: formality is not a suitable engineering goal. While it is a powerful means for achieving engineering goals, such as reliability, it is inappropriate as an end in itself. There is simply no inherent value to the customer in formality, only in the other characteristics that may, or may not, be best achieved by formal methods. This hypothesis has influenced our criteria for specifications and proofs.

Consider a specification S and an implementation I of S. We say that S is "complete and precise enough" if the users of I can determine the service offered by I from S — without referring to I's source code. This criterion focuses on the role of specifications as a communication medium between people. It was chosen to directly support *separation of*

concerns, the key principle for controlling software that is complex and subject to frequent change.

Again consider specification S and implementation I, and a proof that I meets S. We say that the proof is "formal enough" if it convinces the inspection team. As a result, the quality of proofs will depend on the skill, experience and discipline of the inspection team. But this is unavoidable — determining the soundness of a proof will always involve human judgement. From the viewpoint of DeMillo, et al., we are using inspections to change program verification from a formal exercise to a social process aimed at uncovering faults and increasing confidence.

2.2 The Work Products and Their Criteria

Our software development process involves six work products.

The *Requirements Specification* describes the complete observable behavior of the system, including both normal and exceptional behavior, and the changes expected during the lifetime of the system.

The *Module Guide* describes the decomposition of the system into modules and the motivation for the decomposition, in information hiding terms.

There is one *Interface Specification* per module, specifying the assumptions that users of the module are permitted to make about the service it offers.

There is one, possibly more, *Implementation* per module, consisting of code intended to satisfy the interface specification.

The *Test Plan* describes the strategy used for selecting test inputs, and the scheme for applying the tests, including stubs, drivers and test case files.

The *Test Implementation* consists of the code, data and procedures used to implement the test plan.

We have developed criteria for each of these work products. These criteria must be both precise enough to make faults identifiable, and concise enough to be used in practice. Meeting these two goals is challenging. The criteria for specifications emphasize precision and completeness, including both normal and exceptional behavior. Specification documents are written according to standard formats, to make application of the criteria simpler. The implementation criteria are based on three well-known formal constructs, described and illustrated in Section 3. Systematic testing [11], of both individual modules and the full system, is an important part of the approach. We argue, as Fagan does, for the use of both inspections and testing.

We next present the criteria for two work products in detail, as background for the two sections to follow.

2.3 Work Product Criteria — Module Interface Specifications

Our criteria for module interface specifications are shown in Figure 1. Each work product criteria has the five sections shown and may include references to other documents. The *Audience* and *Preconditions* sections name the intended readers and the background knowledge

(a) **Audience**

- Module designer, implementor, tester and user

(b) **Preconditions**

- An understanding of the interface specification language

(c) **Purpose**

- Describe the assumptions users are permitted to make about module behavior, independent of the underlying implementation

(d) **Additional Criteria**

- Describes the interface precisely and completely
- Follows the interface specification language rules
- Satisfies the interface design criteria, where practical
- Is testable: no unjustifiable controllability or observability problems

(e) **References**

- Interface Specification Language Reference Manual
- Papers [12, 11]

Figure 1. Work product criteria — Module Interface Specification

assumed. This information is essential to effective technical communication. For example, it allows the writer to decide which terms and concepts must be defined and which can be safely used without definition. The *Purpose* and *Additional Criteria* sections describe the fundamental purpose and specific constraints, if any, on the work product. The criteria in Figure 1 are defined in terms of three other documents. Our interface specification language is a simple state machine based notation defined in a short (4 page) internal document. We make use of five interface design heuristics, defined and illustrated in a companion paper [12]. Finally, test constraints are defined and illustrated in the third reference [11]. The above criteria establish inspectable requirements aimed at producing module interfaces that are precisely specified in a standard notation, well designed and economically testable.

2.4 Work Product Criteria — Module Implementations

Figure 2 contains our criteria for module implementations, divided into the five standard sections described above. While the criteria (d) are straightforward, the documents named in (d) and (e) require explanation. The Code Format Rules (2 pages) codify traditional requirements for identifier scope and naming, code indenting and commenting. The Code Verification Rules (2 pages) contain a description of the abstraction function, implementation state invariant and proof scheme sketched above. Overall, the implementation criteria establish requirements for code that meets its specification and can be shown to do so, to an inspection team.

2.5 Automated Inspection

While inspections are valuable, they are also labor intensive. Fortunately, many of the more tedious steps can be economically automated. At present we make heavy use of three Unix

(a) **Audience**

- Module implementor, tester

(b) **Preconditions**

- An understanding of the interface specification and implementation languages

(c) **Purpose**

- Provide code whose behavior satisfies the specification

(d) **Additional Criteria**

- Satisfies the Code Format Rules
- Satisfies the Code Verification Rules
- Is testable — no unjustifiable controllability or observability problems

(e) **References**

- Interface Specification Language Reference Manual
- Code Format Rules
- Code Verification Rules
- Papers [11]

Figure 2. Work product criteria — Module Implementation

tools: *lint* for static type checking; *make* to ensure that executables are up-to-date; and *tcov* to evaluate statement coverage during testing. Further automated support is available. For example, considerable labor could be saved through the use of a language with strong type checking and explicit support for modules (we use C at present).

We next present an example of the use of the implementation criteria on a simple stack module.

3 STACK EXAMPLE

Program name	Inputs	Outputs	Exceptions
s_init			
s_push	integer		full
s_pop			empty
g_top		integer	empty
g_depth		integer	

Figure 3. *stack* Interface Specification — Syntax

3.1 Interface Specification

Figure 3 shows the interface specification *syntax* and Figure 4 the *semantics* for the *stack* module, providing a bounded integer stack. The syntax provides the *access program* names

and their parameter and return value types, and the *exception* names. The five sections of the semantics are interpreted as follows. The SPECIFICATION STATE, s, is a varying length sequence of integers. The next three sections describe *stack* behavior in terms of the value of s. Access program calls are divided into two groups: *set calls*, indicated by a s_ prefix, change the specification state; *get calls*, indicated by a g_ prefix, query the specification state. The SET CALL EFFECTS section describes the effect of each set call in terms of changes to s. The GET CALL RETURN VALUES section describes the value returned by each get call, in terms of the value of s. The EXCEPTIONS section defines the calls which exceed the capabilities of *stack*. For example, if s_push is called when `length(s) = SIZ`, the `full` exception must be signaled, and no changes made to s. If g_top is called when `length(s) = 0`, the `empty` exception must be called and g_top may return any integer value. Methods for signaling exceptions vary; one simple approach is shown below in the *stack* implementation.

```
Note:  s_init must be called before any other call.
SPECIFICATION STATE
        s:  sequence [0..]  of integer
SET CALL EFFECTS
        s_init:  s = empty
        s_push(i):  append i to the end of s
        s_pop:  delete the last element of s
GET CALL RETURN VALUES
        g_depth = length(s)
        g_top = s[length(s)-1]
EXCEPTIONS
        s_push(i):  (full:  length(s) = SIZ)
        s_pop:  (empty:  length(s) = 0)
        g_top:  (empty:  length(s) = 0)
CONSTANTS
        SIZ = 100
```

Figure 4. *stack* Interface Specification — semantics

3.2 Implementation Criteria

The implementation criteria are based on three well-known formal constructs.

An *abstraction function* is a mapping from the implementation state — the internal variables of the module — to the specification state — the objects manipulated by the user, as described in the interface specification.

An *implementation state invariant* is an assertion on the implementation state variables that must be true on entry to and exit from the code implementing each call or operation the module provides.

A *loop invariant* is an assertion on selected variables that must be true before and after each execution of the loop body.

Correctness proofs proceed in three steps.

(1) Show that the implementation state invariant is established by `s_init` and is maintained by the other calls.

(2) Argue the absence of certain specific faults, such as variable use before definition, array subscript range error and pointer error.

(3) Argue the correctness of the implementation with respect to its specification, one call or operation at a time, taking the implementation state invariant as given.

Steps (1) and (2) closely follow the scheme presented by Wulf, et al. [13], and based on earlier work by Dijkstra, Dahl, Hoare and others. While steps (2) and (3) are partially redundant, we have found that specifically addressing dangerous faults is quite useful.

3.3 Implementation

A *stack* implementation in C is shown in Figures 5 and 6. The implementation state is as expected. The abstraction function describes the specification state corresponding to each implementation state: `s` is the first `siz` elements of the `stack` array. The implementation state invariant requires that `siz` be consistent with the declared size of `stack` and that the first `siz` elements of `stack` have been given values. We do not require that the abstraction function be defined on every possible value of the implementation state — only on those that satisfy the implementation state invariant.

```
/*** implementation state */
static int stack[SIZ]; /*stack elements*/
static int siz; /*number of elements in stack*/
/*** abstraction function:
*       length(s) = siz and
*       (forall i) if i in [0,siz-1] then s[i] = stack[i]
*/
/*** implementation state invariant:
*       siz in [0,SIZ] and
*       (forall i) if i in [0,siz-1] then
*               stack[i] has been assigned a value
*/
```

Figure 5. *stack* implementation — part I

The code in Figure 6 is unsurprising, except perhaps for the exception processing code. In this implementation, an exception, say `empty`, is signaled by calling the C function `empty`. The *stack* user implements `empty` to perform whatever exception handling is appropriate for his application. So, if `s_push` is called when *stack* is full, `full` is called from `s_push`, which then returns to avoid modifying the implementation state.

```
void s_init() {
        siz = 0;
}
void s_push(x)
int x; {
        if (siz >= SIZ) { full(); return; }
        stack[siz++] = x;
}
void s_pop() {
        if (siz == 0) { empty(); return; }
        --siz;
}
int g_top() {
        if (siz == 0) { empty(); return(0); }
        return(stack[siz-1]);
}
int g_depth() {
        return(siz);
}
```

Figure 6. *stack* implementation — part II

3.4 Verification

In an inspection meeting, review of the *stack* implementation begins with a proof that the implementation state invariant is established by `s_init` (trivial in this case) and is maintained by the other access programs. Consider `s_push`, and assume that the implementation state invariant holds on invocation. There are two cases. If `siz >= SIZ`, then `full` will be signaled and no change made to the implementation state. Otherwise, `siz < SIZ` and `siz` will be incremented, but will still lie in `[0,SIZ]`. The range `[0,siz-1]` does grow to the right, but the new element is assigned a value. So, while `s_push` may change the implementation state, the implementation state invariant will still hold.

Taking the implementation state invariant as given, we next show the absence of certain classes of serious faults. We focus on faults that will cause the module to abort, fail to terminate, or violate C language limitations because these failures are both particularly harmful and difficult to debug. Two classes of faults are relevant to the *stack* implementation: (1) variable use before definition and (2) subscript out of range. We present the proof of (2) for `s_push`. Only one array reference is made: `stack[siz++] = x`. When it is reached, the implementation state invariant and the falsehood of the preceding `if` condition imply that `siz++` is in `[0,SIZ-1]` — the set of legal subscript for `stack`.

The final proof step is to show that each set call modifies the implementation state correctly and that each get call queries it correctly. The underlying basis for set call proofs is shown in Figure 7, adapted from the scheme proposed by Gannon, Hamlet, and Mills [14]. SS denotes the specification state and, for access program P, SP denotes the transformation

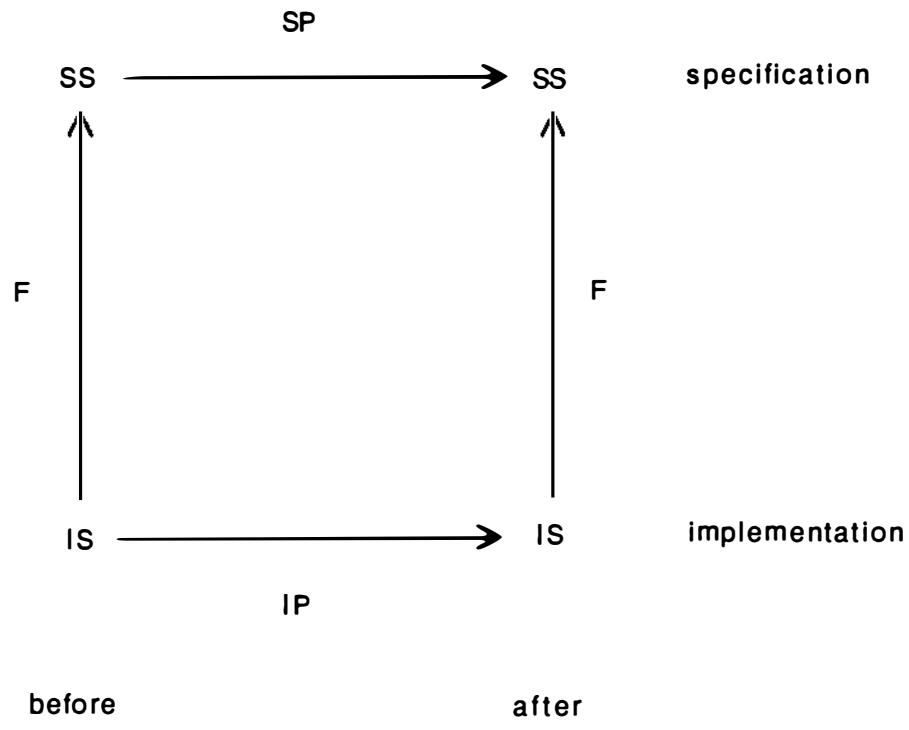


Figure 7. *Commuting diagram*

on SS specified for P, in the interface specification. Similarly, IS denotes the implementation state and IP the implemented transformation on IS — the one that occurs when P's implementation is invoked. F is the abstraction function, showing the SS value corresponding to each IS value. The proof depends on showing that the diagram commutes: for any IS value s, (1) applying F then SP yields the same SS value as (2) applying IP then F. For `s_push(`i`)`, (1) corresponds to taking the first `siz` values of `stack` and appending i and (2) corresponds to assigning i to `stack[siz]` and incrementing `siz`, and then taking the first `siz` values of `stack`. We have found completely formal proofs of set and get call correctness, such as those proposed by Wulf, et al. [13], to be too laborious to be used in practice. Perhaps automated support for theorem proving will change this situation in the future, but in the meantime, we make do with proof sketches.

4 EXPERIENCE AND CONCLUSIONS

4.1 Experience

We have experience with the above approach in academia and industry. At the University of Victoria, we have developed three systems, all in C and on the Unix operating system.

1. BB is a demonstration system consisting of approximately 6,000 lines of source and documentation. BB is a model system: the code and documentation is of the highest quality we have been able to produce, ignoring cost. It is small enough to be easily understood and manipulated, yet large enough to demonstrate modern software engineering techniques. It has become for us the standard example for discussion and initial evaluation of new methods, and for training of new team members.

2. PGMGEN [15, 11] is a module testing tool, consisting of 8,000 lines of source and documentation. Now in its third version, automated module testing has provided significant benefits throughout its develoonsiderable weight was given to testability during interface design. Version 2 testing was done with tests generated by version 1; version 3 tests were generated by version 2.
3. To evaluate our methods in a team environment, a version of the computer game TETRIS was developed as the term project in a third year software engineering course In the resulting 5,000 lines of code and documentation, the requirements were developed by the instructors. All other work products were developed by students, working in six teams of four persons each. For each module, testing and implementation was carried out by different teams. The implementation was given to the test team after a clean compile and an inspection, but before execution, as in the *cleanroom* approach [16]. Module and integration testing went smoothly. Failures were quickly traced to the module at fault, with one exception: a subtle interaction between screen and keyboard input/output that took a half day to discover.

We have now run 44 inspection meetings, consuming approximately 140 person hours in meeting and preparation time. Thus far, we have found as many faults in our work product criteria and inspection methods as we have in the work products themselves. While most of our "proofs" are sketchy, the use of abstraction functions and invariants has been effective. We have found that these constructs often capture key assumptions made by the implementor, but not stated or even consciously known. Perhaps most important, our inspection teams have learned to react negatively to work that is obscure, unnecessarily complex, or poorly documented.

We have research underway aimed at applying our methods to three industrial systems: a CASE tool (55.000 lines/Pascal/MacIntosh), a particle accelerator control system (76,000 lines/C/IBM PC), and a mechanical engineering finite element analysis system (63,000 lines/Fortran/Sun). From our initial efforts, several things have become quite clear.

1. Industrial software development is heavily constrained by the installed base. A practical method must be applicable in a variety of software and hardware environments, and it must be effective even when much of the system was developed with some other approach, or with no systematic approach at all.
2. Often the internal structure of industrial software is undisciplined and undocumented: there is no module guide and there are no interface specifications. Modules are difficult to identify and those that can be found have complex and unnecessary connections with other modules. There is little or no systematic testing, beyond beta testing. There is usually no module testing whatsoever. Because only the system in its entirety is tested, fault isolation is difficult, and compile and link times for test runs are long, especially on microcomputers.
3. Adapting existing systems to our methods, or any other, will be expensive. It will probably be cost effective only on portions of the system that are either extremely problematic or are already being modified for other reasons.
4. Developers of moderate-sized systems (50,000–100,000 lines) have serious software engineering problems. Many of these developers realize that their present methods are ad

hoc and inadequate. This has an important consequence for academics: significant software engineering research can be carried out on inexpensive hardware and with teams of modest size. We do not claim that such research will scale up to 10 million line systems, but only that it could provide valuable benefits to an important group of developers.

4.2 Conclusions

We have presented a hybrid approach to software development where control is achieved through a set of criteria for specification, implementation and test work products. The criteria emphasize precise specification and partial proofs of correctness. While we make frequent use of formalisms, we consciously avoid making formalism a goal.

Although full proofs of correctness still appear impractical, we have found implementation state invariants and abstraction functions to be feasible and beneficial uses of formalism. With modest effort, we have been able to produce these constructs for a variety of modules. In each case we made important discoveries about module behavior. More productive design discussions, and design improvements resulted.

We have also translated the implementation state invariants into C code and used them as dynamic assertions, executed at entry to and exit from each access program. Typically the coding time is small but the execution time penalty is substantial. As a result we normally execute the assertions only during testing.

REFERENCES

[1] M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, July 1976.

[2] D.L. Parnas and P.C. Clements. A rational design process: how and why to fake it. *IEEE Trans. Soft. Eng.*, SE-12(2):251–257, February 1986.

[3] D.M. Hoffman and R. Snodgrass. Trace specifications: methodology and models. *IEEE Trans. Soft. Eng.*, 14(9):1243–1252, 1988.

[4] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[5] R.W. Floyd. Assigning meanings to programs. In *Proc. Symposia in Applied Mathematics (Vol XIX)*, pages 19–32, American Mathematical Society, 1967.

[6] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Inc., 1976.

[7] E.W. Dijkstra. Structured programming. In *Proc. Conf. NATO Science Committee*, 1969.

[8] R.A. DeMillo, R.J. Lipton, and A.J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5), May 1979.

[9] P. Naur. Formalization in program development. *BIT*, 22:437–453, 1982.

[10] R.M. Karp. Reply to 'On the cruelty of really teaching computer science'. *Commun. ACM*, 32(12):1410–1412, December 1989.

[11] D.M. Hoffman. A CASE study in module testing. In *Proc. Conf. Software Maintenance*, pages 100–105, IEEE Computer Society, October 1989.

[12] D.M. Hoffman. On criteria for module interfaces. *IEEE Trans. Soft. Eng.*, 16(5):537–542, May 1990.

[13] W.A. Wulf, R.L. London, and M. Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Trans. Soft. Eng.*, SE-2(4):253–265, 1976.

[14] J.D. Gannon, R.G. Hamlet, and H.D. Mills. Theory of modules. *IEEE Trans. Soft. Eng.*, SE-13(7):820–829, July 1987.

[15] D.M. Hoffman. Hardware testing and software ICs. In *Proc. Pacific Northwest Software Quality Conf.*, pages 231–244, 1989.

[16] R.W. Selby, V. Basili, and F. Baker. Cleanroom software development: an empirical evaluation. *IEEE Trans. Soft. Eng.*, SE-13(9):1027–1038, November 1987.

A PARADIGM FOR TESTING EMBEDDED EXPERT SYSTEMS

Regina Palmer
Martin Marietta Space Systems Company
P. O. Box 179, M/S H4330
Denver, CO 80201
(303) 971-6754

1. INTRODUCTION

This paper addresses the testing that was performed on a power management system program, Space Station Module/Power Management and Distribution (SSM/PMAD), for George C. Marshall Space Flight Center (MSFC), Huntsville, AL. The problem to be solved on this contract was to build a system containing both hardware and software components to automate a Space Station Freedom-like power system. The answer to the power system problem consisted of the following requirements:

1) remotely operable switches that control power to loads

2) ability to produce a schedule of activities using power as a resource and have it be executed by the power system

3) ability to detect, diagnose and recover from power system failures

4) ability to operate autonomously

This paper provides a "quality" perspective of the SSM/PMAD system developed to implement the above problem. Section 2 describes the power system domain and SSM/PMAD system in more detail. Section 3 discusses how testing was performed on the system. Finally, section 4 provides a brief summary of the paper and conclusions.

2. POWER SYSTEM DOMAIN AND SSM/PMAD SYSTEM

When developing this testbed, there were some questions asked. Why? The answer was to provide flexibility and power management to a Space Station type situation. What kind of protection did the system need? Safeguards were needed to protect the system such as smart switches which could trip before there was damage to the system. What about recovery? A power system needed to be able to safely recover from faults.

The operation of the system is controlled by the SSM/PMAD interface. A schedule is created by MAESTRO, converted by FELES (Front End Load Enable Scheduler) to a list of switch commands, and is made available to the SSM/PMAD interface. The schedule of switch events, along with a priority list from LPLMS (Load Priority List Management System), is given to the LLPs (Lowest Level Processors). The LLPs then command switches on and off via the SIC (Switchgear Interface Controller). This architecture is organized as shown in Figure 1 where the LLPs communicate to the hardware.

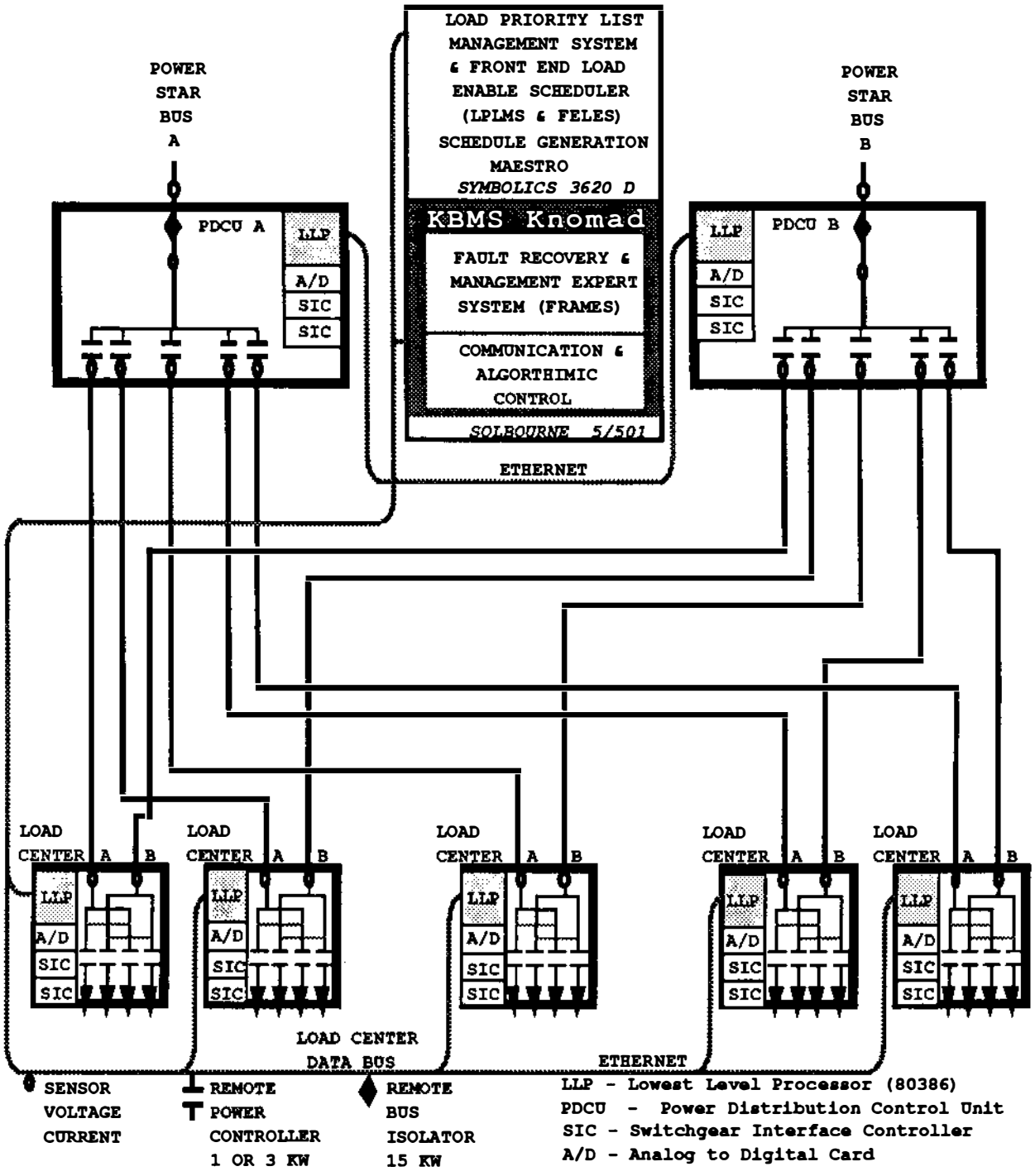


Figure 1 SSM/PMAD System

The SSM/PMAD system provides hardware consisting of smart switches and power distribution. It provides an operational framework at three fundamental levels:[1] The first level is the switchgear (power system) comprised totally of hardware. The second level consists of software components that provide deterministic algorithms for the SSM/PMAD system. The third level is a set of functional software components that provide high-level intelligent control of the system.

The first level consists of switches which are one kilowatt, three kilowatt, and 15 kilowatt. The Remote Bus Isolator (RBI) is a simple switch that can be remotely commanded on and off but will not trip. The 1K and 3K switches are Remote Power Controllers (RPC) that are smart switches and will trip on a low-impedance short. The switch controller (Generic Controller, GC card) adds further logic to the switches. The RPC will detect high-impedance shorts that will result in multiple switches tripping as well as low voltage situations that result in an under voltage trip. Sensors are implemented in different sizes depending on the rating of the current. Current sensors are 50 or 100 amps. Voltage sensors are 120 volt DC sensors. The Analog to Digital (A/D) card collects analog data from the sensors. The Switchgear Interface Controller (SIC) allows communications between the hardware and upper level software components. There is one SIC for each bus of a load center and one SIC for each Power Distribution Control Unit (PDCU).

The second level functions are implemented by the LLPs. Each load center or PDCU is operated by a different LLP. This allows the power system to be more robust since if part of the power system fails, the rest can continue operating. The software architecture for levels two and three is shown in Figure 2.

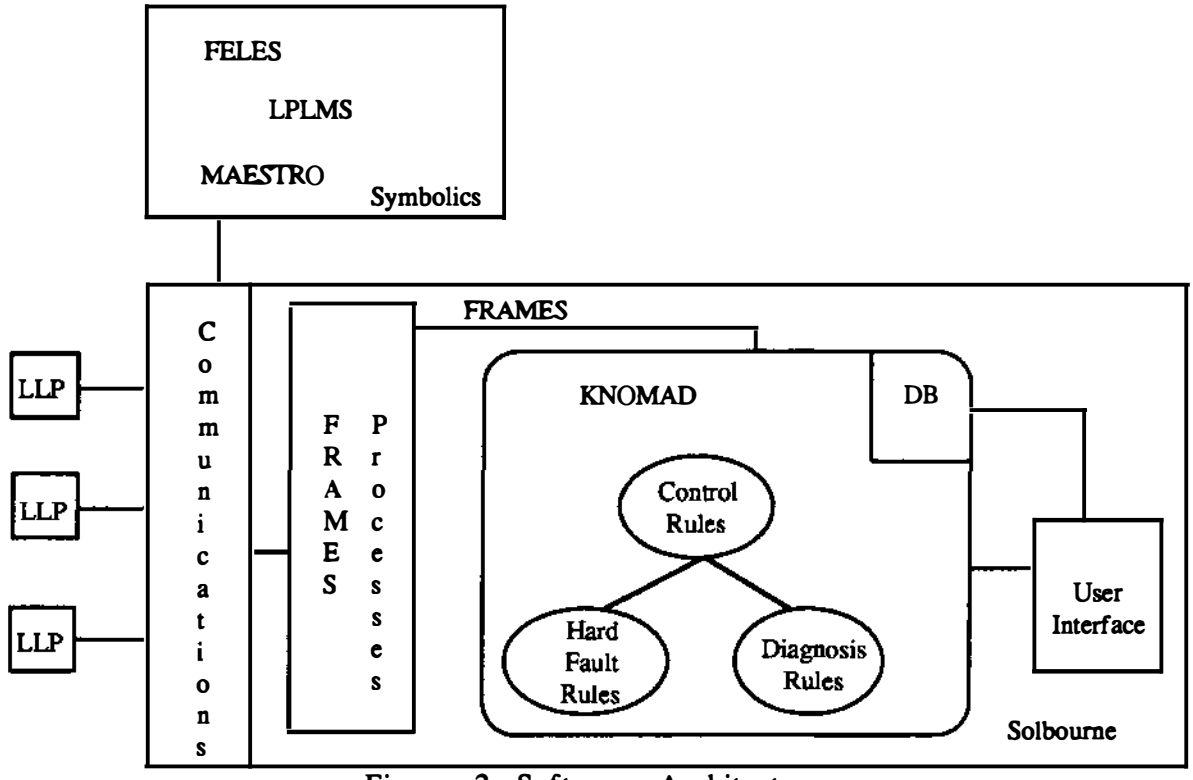


Figure 2 Software Architecture

LLPs are lowest level processors that control switching operations, schedule execution, fault isolation, and performance monitoring. The LLP software command switches within a load center or PDCU by issuing an on or off command for a given switch according to the scheduled switch command events. The LLP software monitors performance of the switches and sensors it controls. It keeps track of the maximum and minimum current readings for each switch as well as the data for all current, voltage and temperature sensors. The schedule passed to the LLP contains maximum and minimum current limits for a load. If the load is pulling more current than is allowed, the LLP will shed the load and inform the

SSM/PMAD interface. If the LLP is a PDCU and a switch is reading out of current limits, the LLP software will not shed the switch, but will inform the SSM/PMAD interface.

The third level consists of knowledge-based functions which include KNOMAD--the Knowledge Management Design System, FELES--the Front End Load Enable Scheduler, FRAMES--the Fault Recovery and Management Expert System, LPLMS--the Load Priority List Management System, and MAESTRO--the scheduler. This level of the system contains the expert systems. An expert system is defined as a computer program that reasons with domain-specific knowledge that is symbolic as well as numerical, uses domain-specific methods that are heuristic as well as following procedures that are algorithmic, performs well in its problem area, explains or makes understandable both what it knows and the reasons for its answers, and retains flexibility[2]. Also included in the third level is other conventional software consisting of user interface and communications software.

KNOMAD is a knowledge management system that consists of two primary levels which are rule management and data management. The data management layer stores data for transferring and sharing with the other components. This layer consists of two levels. The first level is a database used for storing data, and the second level is the interface to the database. The rule management layer is a knowledge base management system and a knowledge base is defined here. A knowledge base in KNOMAD consists of the name of the knowledge base, the rule groups (a set of rules that are related to one another in some fashion) of the knowledge base, which rule groups begin execution, and a termination condition.[3]

FRAMES consists of fault diagnosis, isolation and recovery. FRAMES utilizes a model of the power system for monitoring the activities that are scheduled to occur and what is actually happening in the power system. FRAMES maintains the system status and functions as a watch-dog over the entire power system. Fault diagnosis is performed by collecting data from the fault detection function and performing isolation to determine a diagnosis. Fault isolation is performed by generating a list of switch commands for fault isolation and using the results from commanding the switches to help with the fault diagnosis. Fault recovery, performed after fault diagnosis, is used to inform the scheduler of switches that are no longer available in the power system and cause rescheduling to take place.

Another component of the knowledge-based software is LPLMS. LPLMS initializes and manages the priorities of loads. Initial priorities for power loads will change with occurrence of various system events such as changing availability of system power, passage of time, emergencies, etc. These priorities are managed and allocated in proper ways to assure dependable system performance by LPLMS. For example, in the event of an immediate power change where the scheduler does not have time to compute a new schedule, the LLPs must shed the lowest priority loads to stay within the new amount of power. The priorities are used by the LLPs in the event of anomalous situations and determining which loads not to shed.

Still another component of the knowledge-based software is FELES. FELES interfaces to the scheduler and manages data translations between power system schedules of switch operating characteristics to activity description performances.

The scheduler, MAESTRO, schedules activities that need power. A schedule is a set of activities describing tasks to be performed. The goal of scheduling is to map out when a set of tasks may be completed while making efficient use of available resources. It contains basic knowledge of the overall power system and the required heuristics to ensure good allocation of resources. It is initiated by the user and performed whenever there is an unforeseen resource change in the power system. An activity editor is used to create activities that describe tasks to be scheduled. For example, the editor requests the name of the activity, the priority of the activity ranging from 0-highest to 3-lowest, the duration of the task, etc.

Also in the third level are the user interface and communications software. The user interface software displays the state of the power system to the user and switch and sensor data. The user interface also allows the user to initialize and operate the power system. The user interface is a menu driven system that has a graphical representation of the SSM/PMAD power system (Figure 3).

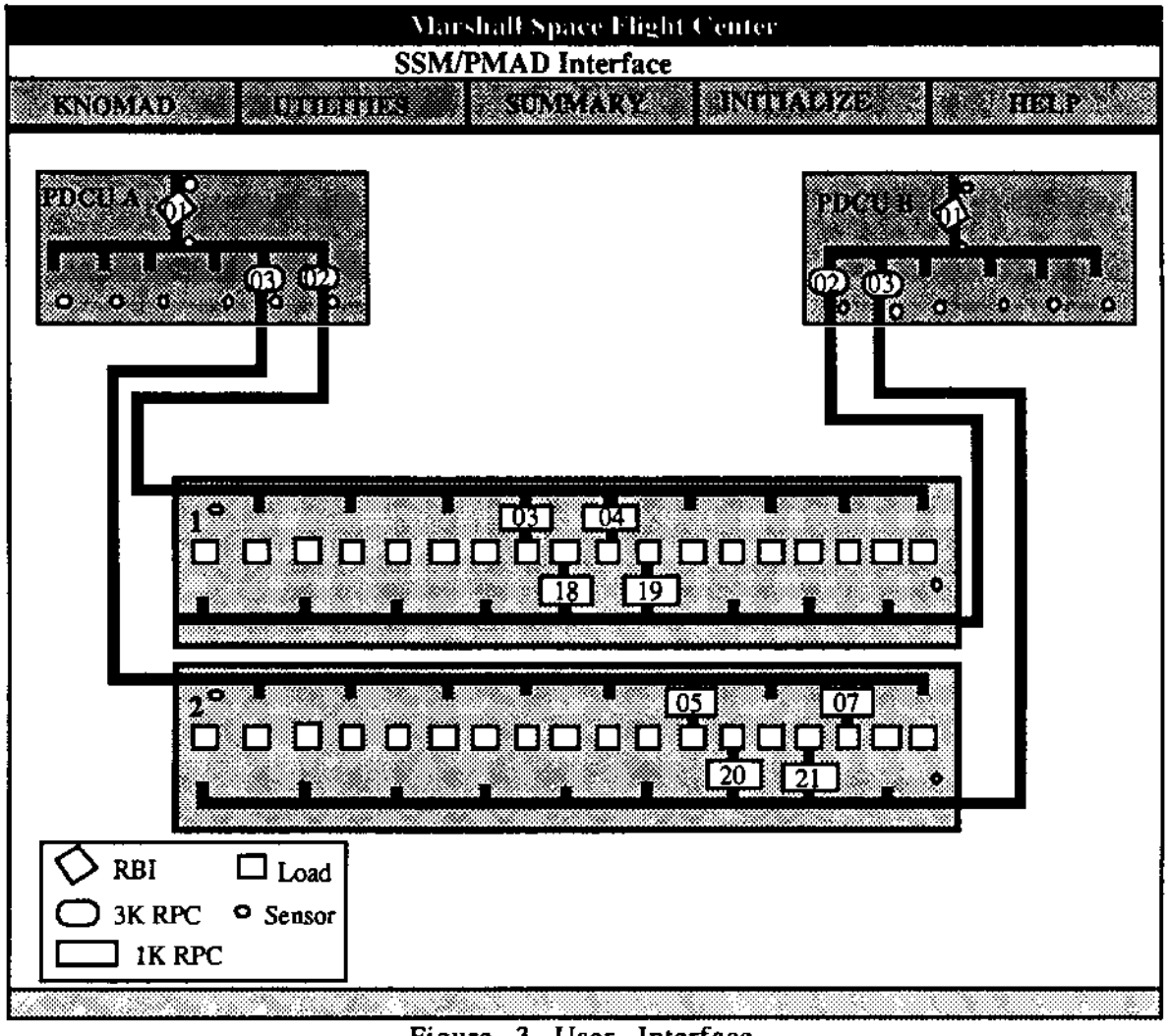


Figure 3 User Interface

The user interface displays the data of selected switches and sensors when requested by the user. An example of the switch and sensor data received is shown in Figure 4.

1K-RPC	
Current:	0.3
State:	Closed
Tripped:	NIL
Powered:	T
Available:	T

Figure 4 Switch Data

AM6	
Current:	0.0
Voltage:	128
Power:	121
Temperature:	NIL

Sensor Data

The communications software consists of the functions necessary for managing the logical connections to the other computers and for describing, transmitting and receiving the data transactions sent between the computers. Transactions are used to send switch data from the power system to the SSM/PMAD interface, for example.

3. TESTING METHODOLOGY FOR THE SYSTEM

The testing of the SSM/PMAD system includes unit, integration, and system testing. Testing of the SSM/PMAD is complex because it contains expert systems with object oriented programming, conventional software, and hardware. The testing methodology of SSM/PMAD consists of testing in an environment that has general requirements instead of specific requirements that are usually found in most systems. The requirements consisted of defining an automation approach for the hardware consisting of possible primary distribution types, possible housekeeping loads and possible user requirements. More detailed requirements consisted of developing a fault management expert system (FRAMES) such that automatic corrective actions were possible (i.e., fault detection and recovery), communications between the hardware and the interfaces, and a load priority management system to define priorities of switches. The software architecture shown in Figure 2 is a result of the software requirements derived from the general requirements.

Unit Testing

Unit testing of conventional software is different from unit testing of expert systems. Unit testing of conventional software consists of testing a specified value and receiving the same result each time; while unit testing of expert system software consists of testing the consistency and completeness of the knowledge base, given the limits of the knowledge engineering. Testing of the integrated knowledge-based and deterministic software yields behavioral results for the entire system. It also provides a measure of the boundaries to which the "system expertise" may be expected to perform.

The conventional software was broken up into needed units such as a unit for the switching command, a unit for receiving data from the interface, a unit for communications between the LLP and the interface, and a unit for sending data. The unit testing of the conventional software consisted of testing each path for expected results and obtaining those results each time. For the LLPs, the test data acquisition routines were tested, and communication to each of the eight LLPs were tested. When an error occurred during testing, a module by module check was performed to detect the module that had generated the error. The LLPs did not have localized responses coming back; therefore, the communications software

had to be examined and based on the data from the communications software, the programmer would know what needed to be changed in the LLP software. Communication from the LLP to the hardware is dedicated.

The expert system software was broken up into rule groups. For example, there were three rule groups for the multiple faults expert system. The first rule group controlled the invocation of the other two. The second rule group was the multiple fault rule group which actually determined a diagnosis. The third rule group was a diagnosis rule group which displayed the diagnosis to the user.

The process for selecting rules for test consisted of matching rules with data in the database, and from the rules that are matched, select some of them and fire them. This is called the match-select-fire cycle. The match phase consists of matching the left hand sides of rules with data in the database and finding those rules that are satisfied. From the resulting set, usually only one, but possibly more, rule(s) are selected in the select phase. Then, those rules that are selected are fired. Firing a rule means that the statements on its right hand side are asserted into the database. This cycle continues until the termination condition is satisfied.[3]

The unit testing of the expert system software consisted of performing the process described in the previous paragraph. Sample test cases were run against the rule groups and experts reviewed the results to assure the accuracy of the system. For example, the symptoms of a "no power to the bus" diagnosis would be that the voltage of the top sensor in the hierarchical group (sensor for RBI 01 on PDCU A in Figure 3) registered 0 voltage and the 3K RPC tripped on undervoltage. The rule would identify to the user that there was less than the nominal voltage supplied to the bus. Therefore it is likely that there is no power to the bus. The testing of the expert system relied heavily on inspection rather than test. Only 50% of the software was tested at the contractor site due to the lack of hardware. When the delivery was made to the customer site where the majority of the hardware resides, additional testing was performed.

Integration and System Testing

Integration testing of the conventional software and expert system software included verifying content and knowledge of the system. Transactions from LLP to switchgear were tested to ensure that the transactions were accurate and that they made sense. Transactions from LLPs to FRAMES were tested to ensure that the communications between the two were working. LLPs were run in conjunction with FRAMES to test that faults were registered properly. Integration testing consisted of testing a piece at a time. FRAMES was tested with the user interface, then communications, then KNOMAD, then the LLPs, etc.

System testing of the SSM/PMAD power system consisted of a demonstration of the system in real-time including scheduling activities to be performed, assuring the activities were scheduled correctly, assuring that the scheduled activities were reflected accurately on the user interface, that faults were detected and reflected correctly on the user interface, and that the system recovered from faults and rescheduled the loads.

Testing of the power system was designed so that a particular diagnosis would be received. For example, the user would inject a fasttrip on a 1K switch to ensure

that the system would diagnose a fasttrip and perform the correct recovery. If a 3K RPC was tripped, then all of the 1K RPCs should also trip.

The system will allow up to eight LLPs to be operated simultaneously. If the hardware goes down for any reason, communication from the LLP to the workstation will attempt to reconnect. If there is a hardware failure and a LLP goes down and communication cannot be re-established between the workstation and the LLP, the LLP must be restarted manually. Buses A and B (Figure 1) are redundant. If one bus gets shut down, the system automatically switches loads to the other bus if scheduled to do so.

An example of a test performed on the SSM/PMAD power system to run a schedule and detect and diagnose faults is documented below. An example of a fault injected and the diagnosis and fault management that occurs will also be described.

1) Initialization is performed on the Solbourne. FRAMES is started and the LLPs initialized. An example of the interface with the switches turned on is shown in Figure 5.

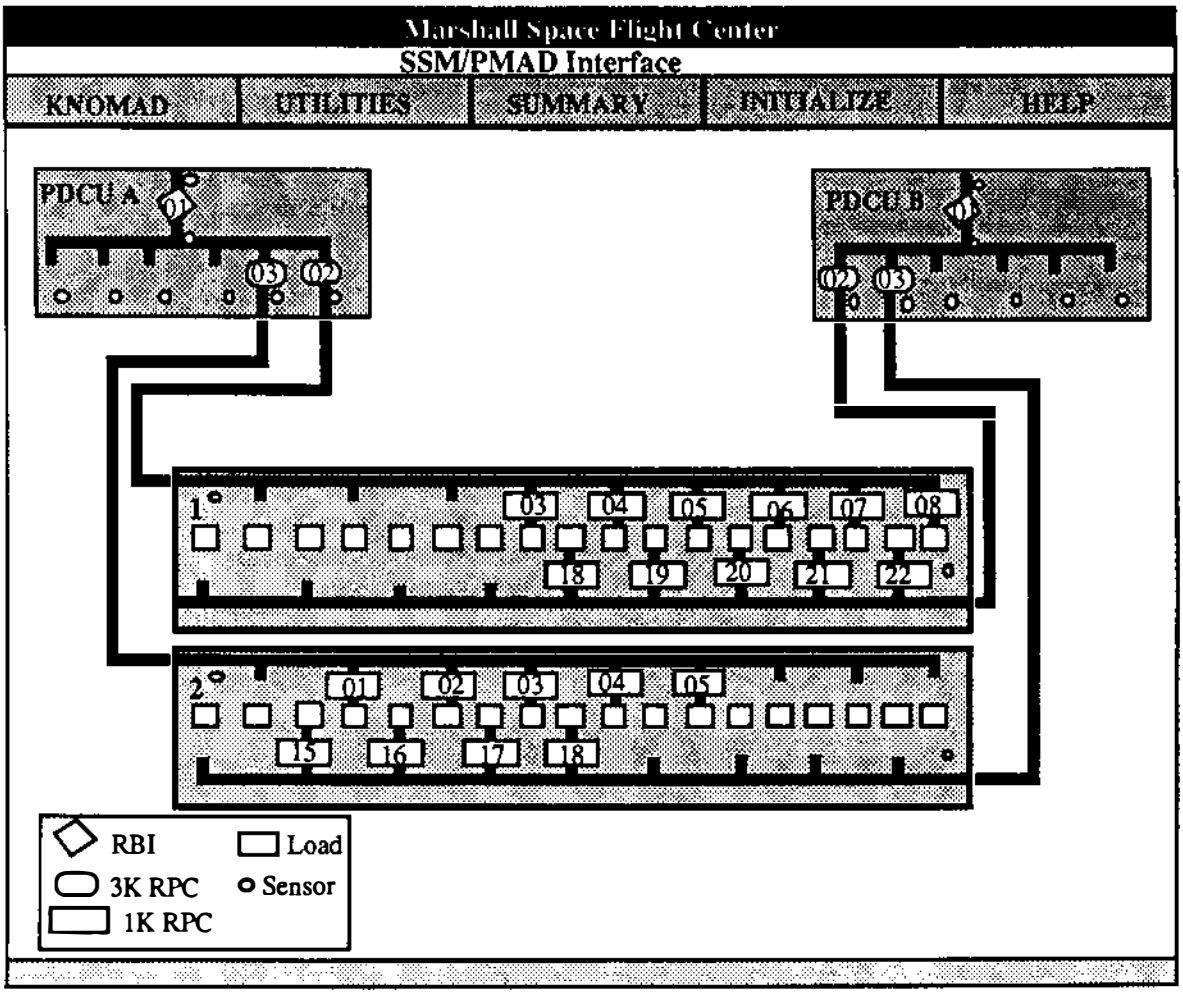


Figure 5 SSM/PMAD Interface with 2 Load Centers

2) A schedule is prepared by MAESTRO and sent to the Solbourne and the LLPs for execution.

3) Switches are turned on and off according to the schedule as prepared by MAESTRO. If a fault occurs, the user interface reflects that status of the system and performs fault diagnosis to determine which switches may no longer be used because of the fault.

4) A Fasttrip on a 3K switch (PDCU B, RPC 03) is injected and the LLP detecting the fault sends a message to the SSM/PMAD interface (Figure 6). FRAMES will in turn request data from each LLP to get a snapshot of the system after a fault has occurred.

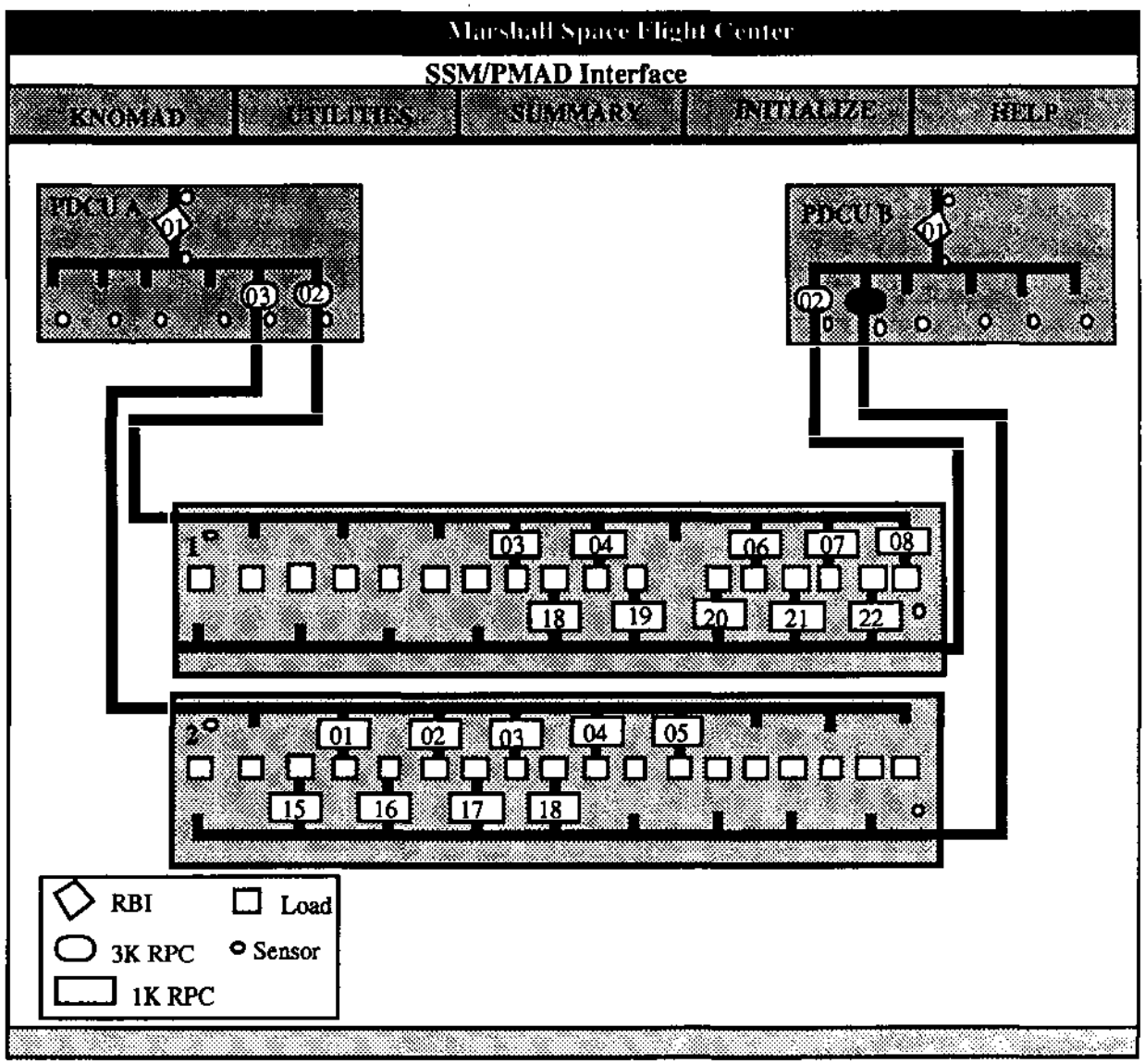


Figure 6 Fasttrip on a 3K Switch

5) After the fault is detected, the LLPs communicate data about their switches and sensors to FRAMES. When FRAMES discovers that there are symptoms indicating a failure in the power system, analysis is used to determine the class of fault that accounts for the symptoms, and further testing is performed to isolate the fault to a particular location in the power system. Once the fault location has been isolated (obtained from switch and sensor information), FRAMES determines and communicates to the scheduler those

switches that are no longer usable. The scheduler uses this information to schedule loads on the remaining usable switches.[1] The user interface is then updated with the current state of the system (Figure 7).

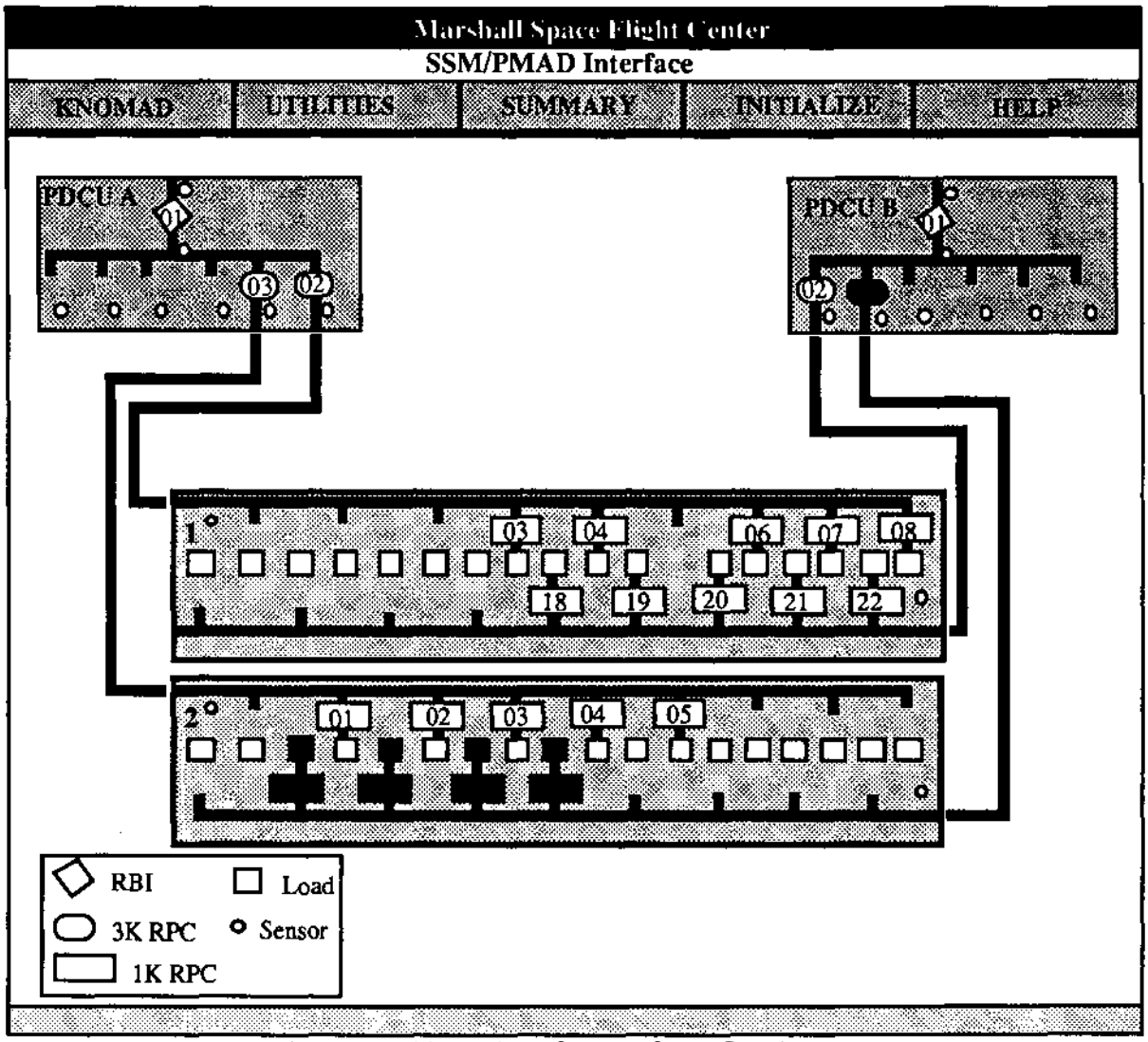


Figure 7 User Interface After Contingency

6) The power system resumes normal operations with the revised schedule.

4. CONCLUSIONS

The power system developed under this contract contains smart switches that control power to loads. It has the ability to produce a schedule of activities using power as a resource and have it be executed by the power system to enable loads. It has the ability to detect, diagnose and recover from power system faults, and it operates autonomously. The software system contains conventional software and knowledge-based software containing components that provide high-level control of the system (expert systems).

The majority of testing of expert systems has been concentrated on validation and verification of expert systems only. There has been little work in integration of experts systems with conventional platforms and then testing them.

The testing method used by this program attempted to treat the expert system software as conventional software by performing unit testing and integration testing prior to the actual system test. An exception to the conventional testing method for unit testing was that all of the paths could not be tested.

Since the start of this program, four deliveries and system tests have been performed using the methodology described in this paper. The latest test and delivery was performed in June 1990. This method has worked consistently and satisfactorily for each of the deliveries and tests.

5. ACKNOWLEDGEMENTS

This work was performed by Martin Marietta Astronautics Group, Space Systems Company under contract number NAS8-36433 to NASA George C. Marshall Space Flight Center, Huntsville, Alabama.

Many thanks to Joel Riedesel who provided the technical support and comments to me during the writing of this paper.

6. REFERENCES

1. Riedesel, Joel D., Chris Myers, and Barry Ashworth, "Intelligent Space Power Automation" in Proceedings of the Fourth IEEE International Symposium on Intelligent Control, 1989.

2. Barr, Avron, Paul R. Cohen, Edward A. Feigenbaum, The Handbook of Artificial Intelligence, Volume IV.

3. Riedesel, Joel D, "Knowledge Management: An Abstraction of Knowledge Base and Database Management Systems" in Proceedings of the Fifth Annual AI Systems in Government Conference, 1990.

Improving the Software Modification Process through Causal Analysis

James S. Collofello
Computer Science Department
Arizona State University
Tempe, AZ 85287
(602) 965–3190

Bakul P. Gosalia
AG Communication Systems Corporation
Phoenix, AZ 85072
(602) 582–7000

Abstract

The development of high quality large-scale software systems within schedule and budget constraints is a formidable software engineering challenge. The modification of these systems to incorporate new and changing capabilities poses an even greater challenge. This modification activity must be performed without adversely affecting the quality of the existing system. Unfortunately, this objective is rarely met. Software modifications often introduce undesirable side-effects leading to reduced quality.

In this paper, the software modification process for a large, evolving real time system is analyzed using causal analysis. Causal analysis is a process for achieving quality improvements via defect prevention. The defect prevention stems from a careful analysis of defects in search of their causes. This paper reports our investigation of several significant modification activities resulting in about two hundred errors. Recommendations for improved software modification and quality assurance processes based on our findings also are presented.

Biographical Sketch

James S. Collofello received his Ph.D. degree in computer science from Northwestern University. He is currently an associate professor of computer science. His research interests include software engineering, software quality assurance and software maintenance. He has researched and consulted extensively over the last ten years in the software engineering and software quality assurance areas with several large companies. He has also written many research articles in this area.

Bakul P. Gosalia received his B.S. degree in material engineering from Indian Institute of Technology, Bombay, India. He is a member of technical staff at AG Communication Systems Corporation; he has wide experience with large real time systems. His research interests include software quality assurance and software maintenance. He is currently working towards a master's degree in computer science at Arizona State University.

Background

The traditional approach to developing a high quality product consists of applying a development methodology with a heavy emphasis on defect detection. These defect detection processes consist of walkthroughs, inspection and various levels of testing. A more effective approach to developing a high quality product is an emphasis on defect prevention. An effective defect prevention approach which is gaining popularity is causal analysis [Gale90, Mays90,Jone85,Phil86]. Causal analysis consists of collecting and analyzing software error data in order to identify their causes. Once the causes are identified, process improvements can be made to prevent future occurrences of the errors.

Most of the research in defect prevention has focused upon development processes. This is unfortunate considering the high percentage of effort consumed by software modification processes. Modifying a large, complex system is a time-consuming and error prone activity. This task becomes more difficult over time as systems grow and their structure deteriorates [Coll87]. Thus, the need for defect prevention is very strong in order to preserve the quality of evolving systems.

This paper describes an attempt to improve the software modification process through causal analysis. In the remaining sections, our data collection and analysis approach will be described as well as our results. Recommendations will also be presented for preventing defects during modification processes.

Data Collection and Analysis Approach

In order to improve the software modification process, error data was collected and analyzed for a version of a large switching system. The size of the system was approximately 2.5 million lines of code. The version analyzed was developed by gradual modifications to the software, firmware and hardware. A software load was developed at regular intervals to integrate the modifications as illustrated in figure 1.

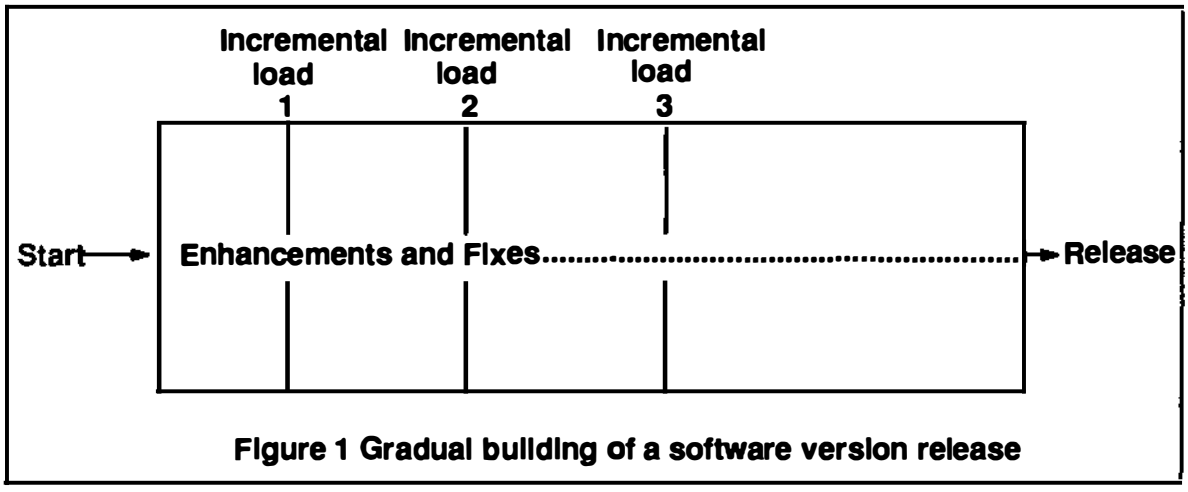


Figure 1 Gradual building of a software version release

Data consisting of problem reports was captured during the first phase of integration testing for each of the incremental loads. This phase verifies the basic stability and functionality of the system

prior to testing the operation of new features, functions, and fixes. Problem reports were analyzed during the testing of each on the 11 incremental loads.

Our analysis of problem reports involved an error prioritization, an error categorization and an error causal analysis.

Errors were prioritized into 3 categories.

1. Critical
2. Major
3. Minor

Critical errors impair functionality and prohibit further integration testing. Major errors partially impair functionality. Minor errors do not affect normal system operation.

Errors were also grouped into error categories based on the nature of the problem. The error categories are described below:

1. **Design Error:** The design error category reflects software errors caused by improper translation of requirements into design during modification. The design at all levels of the program and data structure is included. The following types of design errors are typical:

    1. Logic error
    2. Uninitialized variable
    3. Computation error
    4. Cases missing
    5. External conditions not considered
    6. Timing errors
    7. Data design errors
    8. Errors in I/O concepts
    9. Error conditions not handled

2. **Incompatible Interface:** Incompatible interface errors occur when the interfaces between two or more different types of modified components were not compatible. The following are common types of incompatible interface errors:

    1. Different parameters passed to the procedures/functions than those expected
    2. Different operations performed by the called modules, functions, or procedures than expected by the calling modules, procedures or functions
    3. Code and database mismatched
    4. Executive routines and other routines mismatched
    5. Software and hardware mismatched
    6. Software and firmware mismatched

3. **Incorrect Code Synchronization from Parallel Projects:** For a large evolving system, software development cycles of multiple releases overlap as shown in figure 2.

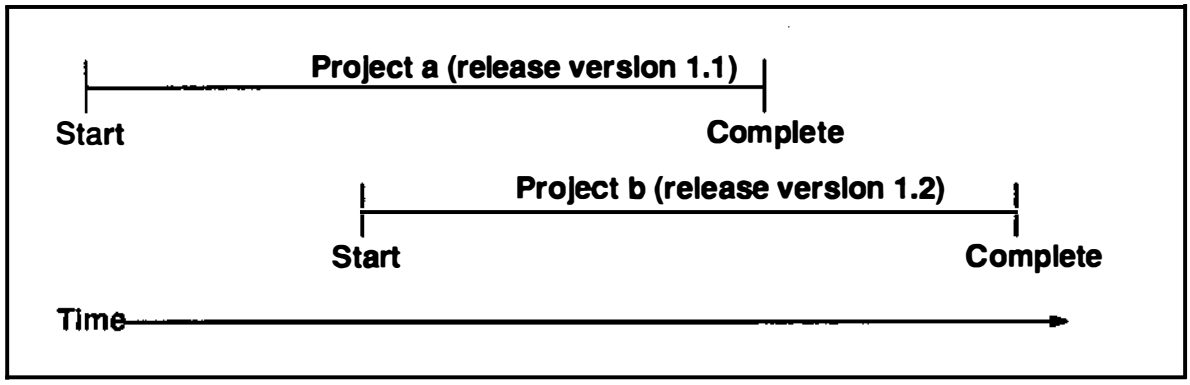


Figure 2 Software development of large evolving systems.

This category of error occurs when changes from project a, the previous project, are incorrectly carried into project b, the current project.

4. **Incorrect Object Patch Carryover:** During integration testing, object patches are sometimes needed when there is no time or resources for re-compiling and re-linking. These object patches may become a part of later incremental loads. Errors committed while modifying modules with object patches fall into this category.

5. **System Resource Exhaustion:** This error occurs when the system resources (such as memory and real-time) become insufficient.

Finally, errors were analyzed to identify their underlying causes and grouped into causal categories.

The software designers who introduced the defects were interviewed. The interviews were informal and performed very carefully so as not to make the designers defensive. The purpose of the analysis (error prevention) was made clear prior to the interview.

The following sets of questions were asked to identify the root causes of the errors:

1. Why did the error occur?
2. Did the error occur due to a lack of communication?
3. Was the error caused by problems in the existing process or methodology?
4. Did the error occur due to a lack of knowledge or training?
5. Were there any deviations from the standard process/methodology ?

The error causal categories are described below:

1. **System Knowledge / Experience:** This causal category reflects the lack of software design knowledge about the product or the modification process. Some examples include:

1. The developer did not understand the purpose of a new function.
2. The developer did not understand the existing design.
3. The developer did not understand the modification process.

2. **Communication:** This causal category reflects communication problems concerning modifications. It identifies those causes which are not attributed to a lack of knowledge or experience, but instead to incorrect or incomplete communication.

An example is the failure of a design group to communicate last-minute modifications to the others who had interfacing code.

3. **Software Impacts:** This causal category reflects software developer failure to consider all possible impacts of a software modification.

4. **Methodologies/Standards:** This causal category reflects methodology and/or standards violations. It also includes limitations to existing methodologies or standards which contributed to the errors.

5. **Feature Deployment:** This causal category reflects problems caused by the failure of software, hardware, firmware or database components to integrate correctly. It is characterized by a lack of contingency plans to prevent problems caused by missing components.

6. **Supporting Tools:** This causal category reflects problems with supporting tools which introduce errors. An example is an incorrect compiler which introduces errors in object code.

7. **Human Error:** This causal category reflects human errors made in the modification process which are not attributable to other sources.

Data Collection Results

The main results of our investigation are presented in this section. Figure 3 presents an overall distribution of error categories across all the problems. Figures 4 through 6 present distributions of error categories across each of the problem priorities. Figure 7 presents the distribution of causal categories for all the problems.

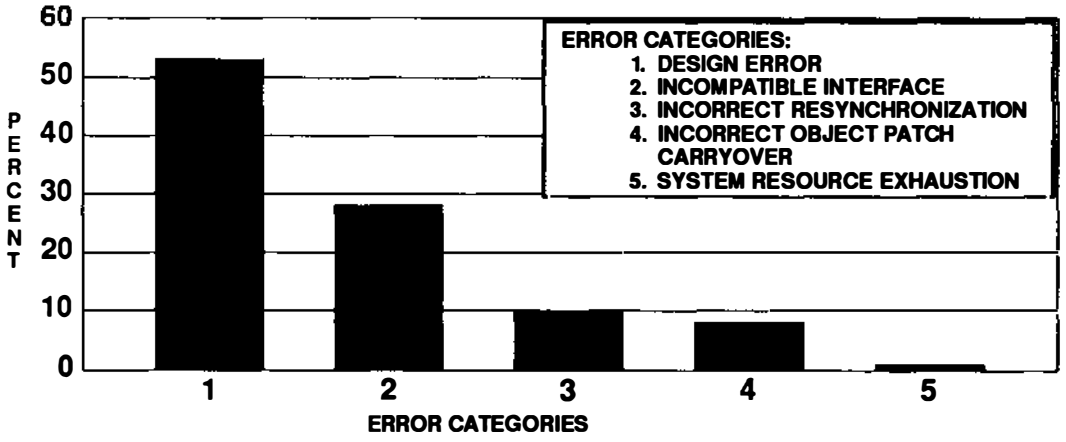


Figure 3 Error category distribution of all the problems

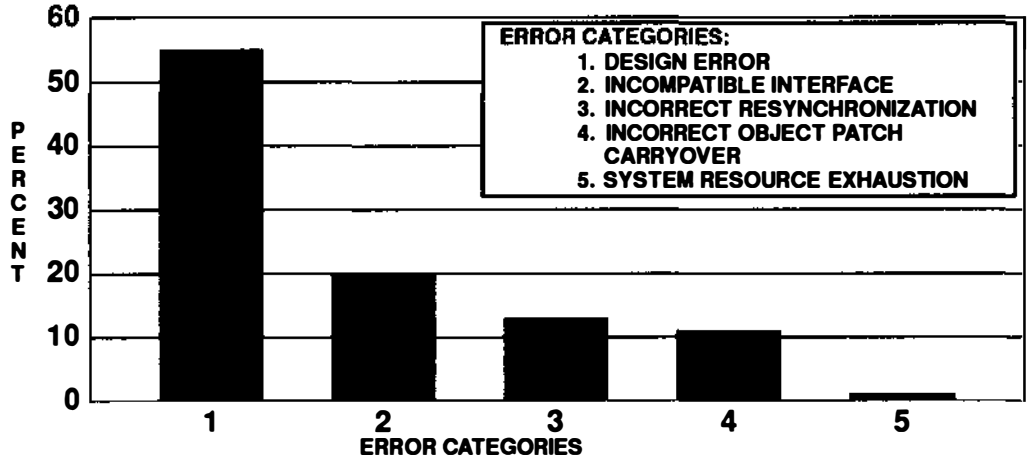


Figure 4 Error category distribution of critical problems

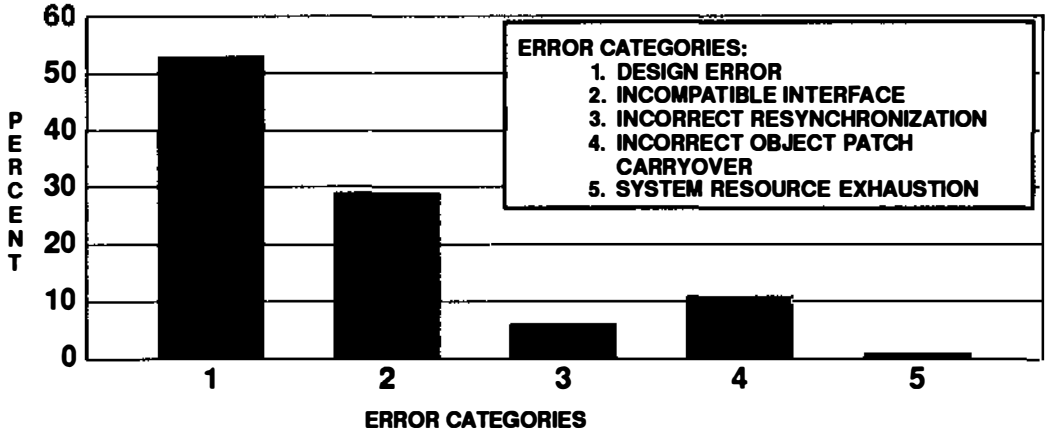


Figure 5 Error category distribution of major problems

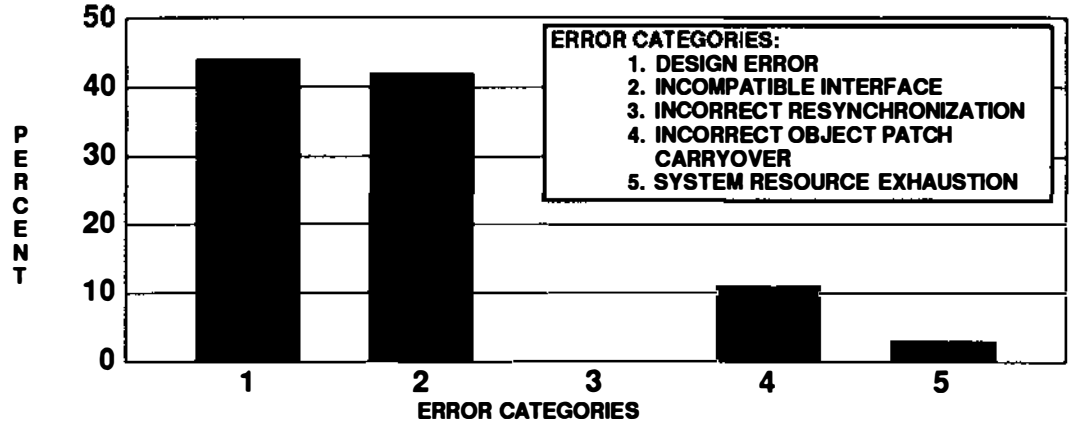


Figure 6 Error category distribution of minor problems

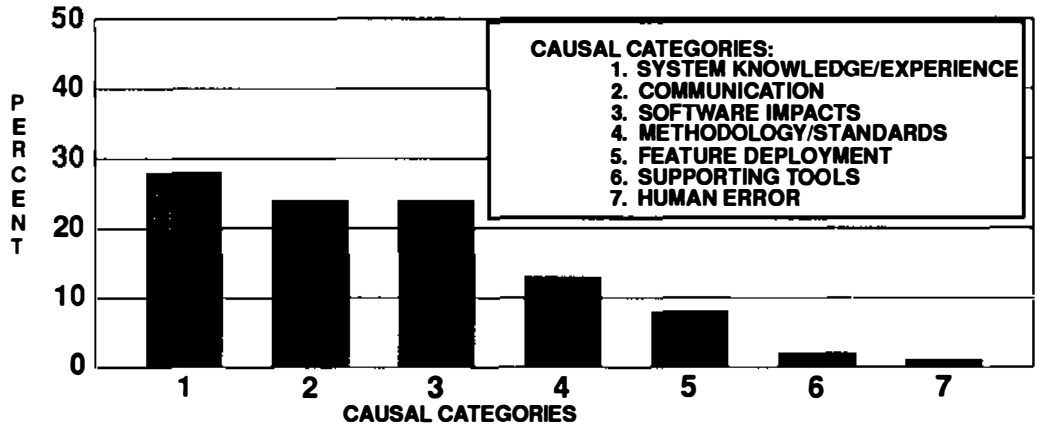


Figure 7 Causal categories distribution of all the problems

Implications and Recommendations

Based upon the data collected, the following recommendations can be made for organizations performing modifications to large evolving products.

1. **Increase experience level:** Twenty eight percent of all problems were caused by insufficient product knowledge. Efforts must be made to keep experienced maintenance personnel on the project and increase training for new personnel.

2. **Improve communication:** Twenty four percent of all problems were caused by insufficient or incorrect communication among the players in the software modification process. This suggests the need for formal methodologies for modification processes which describe documentation procedures and review processes.

3. **Identify software impacts:** Twenty four percent of all problems were caused by failing to consider all of the impacts of modifying the existing code. Several possibilities exist for improvement in this area. Tools which help identify impacts are an optimal solution. Checklists to be followed in reviews of changes can also help in identifying potential impacts.

4. **Enhance Methodologies/Standards:** Thirteen percent of all problems were caused by methodology and/or standard violations or limitations. Continuous enhancements of methodology and standards as the system evolves and more rigorous follow up of methodologies and standards are required.

5. **Develop feature deployment plan:** Eight percent of all problems were caused by the lack of feature deployment plans. Feature deployment plan should be developed during the high level design of a feature requiring major modifications in software, firmware, hardware, and database components. The feature deployment plan should indicate the convergence of software, firmware, hardware, and database modifications and outline contingency plans if the schedules are changed.

Conclusions and Future Research

This paper has helped focus attention on ways for improving the quality of software modification processes. Additional research needs to be performed to assess whether the results of this study are representative of that experienced in other large evolving systems. Other recommendations must also be considered for improving modification processes and all recommendations need to be evaluated in terms of their return on investment.

References

COLL87 J. Collofello and J. Buck, "Measuring and Managing Software Maintenance", IEEE Software, No. 5, pp. 46-51, Sept. 1987.

GALE90 J. L. Gale, J. R. Triso and C. A. Burchfield ", Implementing the Defect Prevention Process in the MVS Interactive Programming Organization", IBM Systems Journal, Vol.30, No. 1, pp.33-43, 1990.

JONE85 C. L. Jones, "A Process–Integrated Approach to Defect Prevention", IBM Systems Journal, Vol.24, No. 2,pp. 50-164, 1985.

MAYS90 R. G. Mays, C. L. Jones, G. J. Holloway and D. P. Studinski ",Experiences with Defect Prevention",IBM Systems Journal, Vol.30, No. 1,pp.4-32, 1990.

PHIL86 R.T.Philips," An Approach to Software Causal Analysis and Defect Extinction", IEEE Globecom 1986 1, No. 12, pp. 412-416, Dec.1986.

A Quality Assessment Framework for Ada-Based Designs

Diane Mularz
The MITRE Corporation
7525 Colshire Dr., MS: Z645
McLean, VA. 22102, U.S.A.
703-883-5598
e-mail: mularz@mitre.org

William Evanco
The MITRE Corporation
7525 Colshire Dr., MS: Z645
McLean, VA. 22102, U.S.A.
703-883-6102
e-mail: evanco@mitre.org

Roberta Hutchison
The MITRE Corporation
7525 Colshire Dr., MS: Z645
McLean, VA. 22102, U.S.A.
703-883-7037
e-mail: hutchiso@mitre.org

Francis X. Maginnis
The MITRE Corporation
7525 Colshire Dr., MS: Z666
McLean, VA. 22102, U.S.A.
703-883-6019
e-mail: maginnis@mitre.org

Douglas W. Cooke
The MITRE Corporation
7525 Colshire Dr., MS: Z676
McLean, VA. 22102, U.S.A.
703-883-5385
e-mail: cooke@mitre.org

Abstract

Determining the quality of a software design requires a definition of quality and an assessment approach that can be consistently applied to many applications. This paper presents preliminary work done to develop a framework for assessment of designs based on software engineering principles and it describes an approach to use in performing an assessment. The framework is based on a fundamental premise of software engineering that systematic application of such principles as modularity, information hiding, and abstraction during design will lead to high quality software, i.e., software that exhibits desirable characteristics such as maintainability and reliability. Initial efforts in defining the framework were oriented toward applications for which design information is available in an Ada Program Design Language (PDL) or code. This orientation reflects the trend towards use of machine-readable, compilable, representations of a design rather than paper documentation. The approach includes methods for characterizing a system to structure the analysis, identifying the important software engineering principles that should be reflected in the design, and assessing the design for indications of the use of those principles. Measures that can serve as indicators of the use of the principles are presented.

Suggested keywords: Ada, design assessment, design quality, software engineering, software engineering principles, design evaluation, software quality.

1. Introduction

Determining the quality of a software design requires a definition of quality and an assessment approach that can be consistently applied to many applications. This paper presents preliminary work to develop a framework for assessment of designs based on software engineering principles and it describes an approach to use in performing an assessment.

The need for a more formal design assessment approach is becoming increasingly evident as software systems increase in scale and complexity. In particular, designs for large government systems require assessment at times such as contractor selections, in progress design reviews, and prior to future enhancement decisions. Expected benefits of applying assessment techniques such as defined in this work include:

- Clarification of software engineering objectives — A standard approach to design assessment may help clarify what is meant by design quality. Studies have shown that most product defects are introduced in the design stage (Boehm, 1981). Therefore, "software design represents the strongest opportunity for improving software quality" (Grady and Caswell, 1986).
- Effective use of resources — À methodology should not have to be re-invented for each new design assessment effort. Instead, each assessment can make use of a standard methodology, which is refined as collective experience grows. Tools to support a well-defined methodology can be more readily identified, made available, and supported if precise needs are known.
- Improved quality of results — Assessments currently tend to be ad hoc. A design assessment methodology can make the process more rigorous and objective as well as repeatable. A consistent methodology may also simplify the process of capturing experiences and provide better data collection for future software engineering analyses. Consistency of assessment products allows for concentration on report contents rather than structure.

The approach given here includes methods for characterizing a system to structure the analysis, identifying the important software engineering principles that should be reflected in the design, and assessing the design for indications of the use of those principles. Many measures were identified that serve as indicators of the use of the principles to assist the assessor in targeting portions of the design for more detailed analysis.

The underlying framework of this approach is based on a fundamental premise of software engineering that systematic application of such principles as modularity, information hiding, and abstraction during design will lead to high quality software, i.e., software that exhibits desirable characteristics such as maintainability and reliability. The methodology examines software products for evidence that commonly accepted software engineering principles were applied in their design. We assume that principles such as abstraction and information hiding are sufficiently general that any "good" design should exhibit them, regardless of the method used to produce the design. Of course, the principles are not absolute; designers must sometimes trade off adherence to the principles against other considerations, most notably functional and performance requirements. However, software quality factors are nevertheless important, and are more easily treated in a general purpose methodology than is functional correctness. Moreover, in our experience, requests for design

assessment support tend to focus on quality factors, and on the degree to which design techniques indicative of correctness and good performance have been used.

Initial efforts in defining the framework were oriented toward applications for which design information is available in an Ada Program Design Language (PDL) or code. This orientation reflects the trend towards use of machine-readable, compilable, representations of a design rather than paper documentation. While Ada-based designs facilitate automation of a design assessment framework, the concepts described here can more generally be applied to other environments.

2. A Design Assessment Framework

The methodology presented in this paper is based on an assessment framework that relates software quality factors, software engineering design principles, views of a software product, design representations, measures, and tools. We will describe the framework as it now stands, but we expect it to continue to evolve as we gain more experience.

The foundation of the framework is a set of quality-related outcomes, or *quality factors*. These factors identify desirable characteristics of delivered software, such as reusability, maintainability, or portability. The ultimate goal of design assessment is to be able to predict, on the basis of the design, the degree to which a software product will exhibit the quality factors (e.g., how efficient the design is, or how maintainable).

The problem is that quality factors can only be measured *directly* after a system has been completely implemented, and in certain cases, only after it has been operational for some time, whereas, a design assessment task requires us to judge a system's quality on the basis of its design. As an example, maintainability, a common concern in a design assessment, can be defined in terms of the effort required to correct defects. But this definition can be used directly to compute a measure of maintainability only after a system has been fielded, when actual data on the number of defects and the effort required to correct them has become available. In our methodology, we attempt to overcome this difficulty by relating quality factors to *software engineering principles*.

It is a fundamental premise of software engineering that systematic application during design of software engineering principles, such as modularity, information hiding, and abstraction will lead to high quality software. An abundance of literature espouses design methods that, if properly applied, will yield designs reflecting these principles. (See [Booch, 1987] and [Meyer, 1988] for two recent contributions.)

If the application of software engineering principles leads to quality products, then it seems intuitive that an assessment of a design for adherence to these principles should provide an indication of the quality of the product. We accept this premise in our framework.*

Software engineering principles, are frequently vaguely defined and not directly measurable. What we can do, however, is identify objective *measures* that are intuitively

*Our long-term goal is to establish an empirical, statistical correlation between software engineering principles and quality factors. Although some work has been done in this area, much remains to be done, particularly for Ada-based designs. For the present, therefore, we appeal primarily to expert opinion to justify our postulated relationships.

related to a software engineering principle. In so doing, we *operationalize* the principle. A measure may not fully capture the concepts connoted by a principle. Measures are indicators, rather than definitions, of the associated principles. Measures can suggest possible problem areas and provide guidance about which portions of a design should be carefully studied, but they cannot, by themselves, demonstrate whether a design is "good" or "bad."

Each measure described in our framework is defined in terms of a subset of the design information created by the designers of a system. This information is termed a design *view*. Views are abstract descriptions of collections of design data; we refer to the concrete forms in which design data are presented (e.g., source code, dataflow graphs) as *design representations*. In our work to date, we have restricted these representations to be Ada-based because we believe that software engineering principles tend to be more structurally explicit in Ada than in other languages. However, our approach could also be extended to other languages.

In the context of the present framework, *tools* support project characterization, allow an assessor to browse through available design data, extract data corresponding to a particular view from one or more design representations, convert one design representation to another, and compute measures from design representations. Tools are essential to the effective application of our methodology because of the volume of design materials that must be assessed:

The following points summarize our design assessment framework:

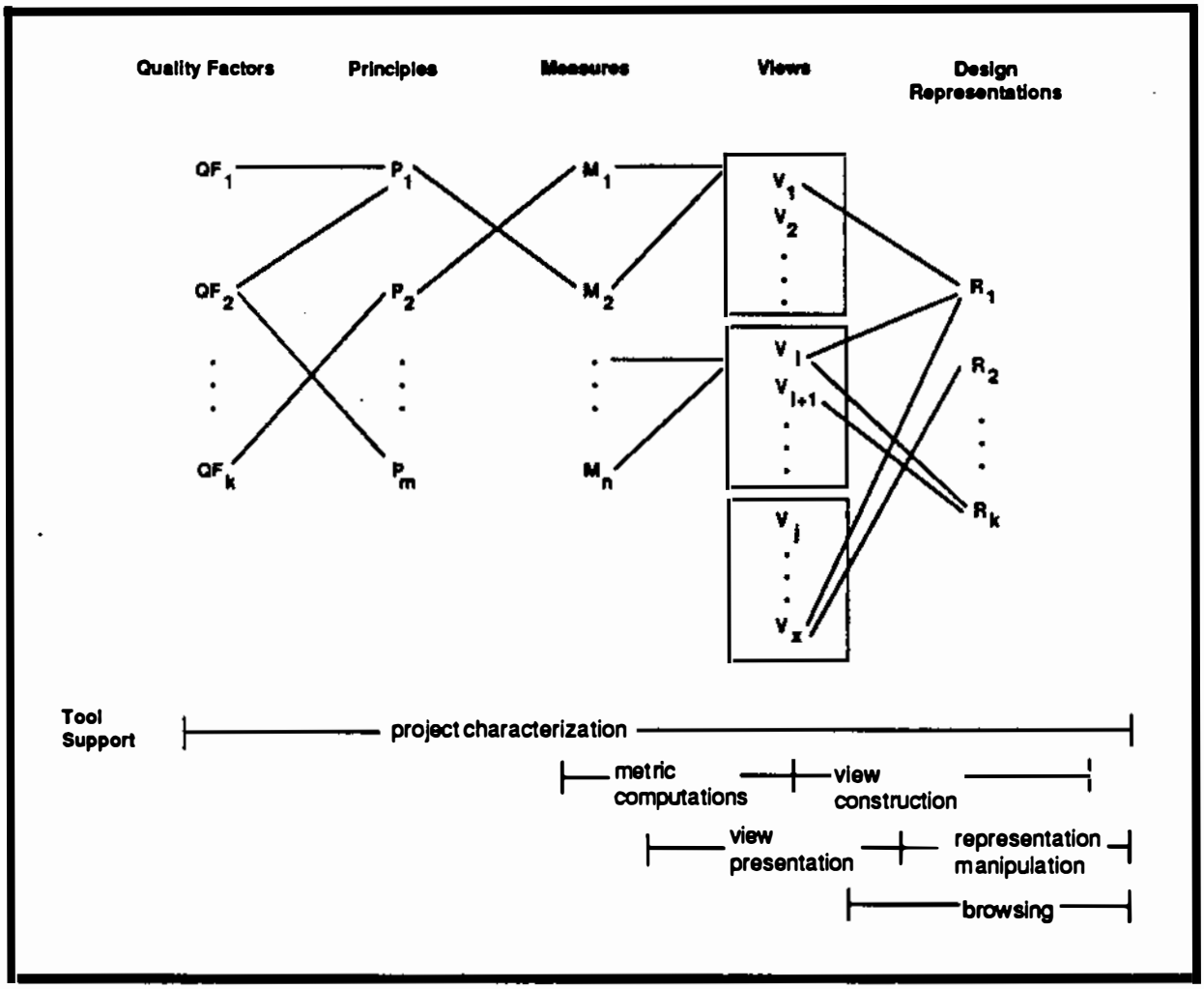
- Quality factors define the desired ultimate characteristics of a software system.
- Software engineering principles define characteristics of a software design that lead to high-quality systems, as measured by the quality factors.
- Measures capture objective, quantifiable characteristics of designs and operationalize the software engineering principles.
- Views and view classes identify the design information needed to calculate the measures.
- Design representations describe the form of the design products from which views are extracted.
- Tools process design representations to compute measures, to convert from one representation to another, to extract and present views, or to provide related support to an assessor.

These relationships are illustrated in figure 1. Note that, in general, the relationships between elements of the framework are many-to-many. For example, several software engineering principles may support a given quality factor, and several quality factors may be supported by a given principle.

2.1 Quality Factors

Quality factors describe desirable characteristics of software. For this framework we have developed a reference set of quality factors based on (ANSI/IEEE-STD-729, 1983; Bowen et al., 1985; and Deutsch, 1988). The reference set is as follows:

Figure 1
Design Assessment Framework



- Correctness — (1) The extent to which software is free from design defects and from coding defects, that is, fault free. (2) The extent to which software meets its specified requirements. (3) The extent to which software meets user expectations.
- Efficiency — The extent to which software performs its intended functions with a minimum consumption of computing resources.
- Expandability — The extent to which the software can be improved for performance or other software attributes.
- Flexibility — The extent to which a software product can be made usable in a changed environment.

- Integrity — The extent to which unauthorized access to, or modification of, software or data can be controlled in a computer system.
- Interoperability — The ability of two or more systems to exchange information and to mutually use the information that has been exchanged.
- Maintainability — The ease with which the software product can be modified to overcome existing faults.
- Portability — The ease with which software can be transferred from one computer system or environment to another.
- Usability — The initial effort required to learn, and the recurring effort to use, the functionality of the software.
- Reliability — (1) The ability of a program to perform a required function under stated conditions for a stated period of time. (2) The probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function not only of the existence of faults in the software, but also of the inputs to the system and of the use of the system. The inputs to the system determine whether existing faults, if any, are encountered.
- Reusability — The extent to which a software module can be used in multiple applications.
- Safety — The absence of unsafe software conditions.
- Survivability — The continuity of reliable software execution (perhaps, with degraded functionality) in the presence of a system failure.
- Verifiability — The extent to which the software design exhibits characteristics affecting the effort to verify its operation and performance.

2.2 Principles

Whereas quality factors define desirable outcomes of a software engineering activity, principles serve as guidance to the software designer. In our framework, principles also serve as motivators for developing assessment measures.

Ross et al., (1975) were among the first to systematically discuss software engineering principles and their relationship to quality factors. For this framework, an initial set of well-defined principles was established based on (ANSI/IEEE-STD-729, 1983; Ross et al., 1975; Meyer, 1988) as follows:

- Abstraction — (1) A view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information. (2) Data abstraction — The result of extracting and retaining only the essential characteristic properties of data by defining specific data types and their associated functional characteristics, thus separating and hiding the representation details. (Note: we define data abstraction here since it is a particular form of abstraction and an important principle in the context of Ada.)
- Completeness — Ensures that all the essentials of an abstraction, for example, are explicit and that nothing essential has been omitted.

- Confirmability — Directs attention to methods for finding out whether stated goals have been achieved. Applied to design issues, conformability refers to the structuring of a system so it can be readily tested.
- Explicit Interfaces — Whenever two modules A and B communicate (i.e., exchange information), this must be obvious from the text of A or B or both.
- Information Hiding — The technique of encapsulating software design decisions in modules in such a way that the module's interfaces reveal as little as possible about the module's inner workings; thus, each module is a "black box" to the other modules in the system. The discipline of information hiding forbids the use of information about a module that is not in the module's interface specification.
- Linguistic Modular Units — The formalism used to express designs, programs, etc., must support the view of modularity. That is, modules must correspond to syntactic units in the language used.
- Minimal Interfaces — Every module should communicate with as few other modules as possible. This restricts the overall number of communication channels between modules in a software architecture.
- Modularity — The extent to which software is composed of discrete components such that a change to one component has minimal impact on other components.
- Open-Closed — A satisfactory modular decomposition should yield modules that are both open and closed. A module is said to be open if it is still available for extension. A module is said to be closed if it is available for use by other modules.
- Weak Coupling — If any two modules communicate at all, they should exchange as little information as possible. This relates to the size of intermodule connections rather than to their number.

We developed a mapping between quality factors and principles by synthesizing mappings presented by Bowen et al., (1985); Meyer, (1988); Ross et al., (1975); and Booch, (1987). This mapping is shown in table 1. Note that three quality factors — efficiency, integrity, and safety — are not mapped. The framework will be extended to include these factors in the future. In our work to date, we have emphasized three of these principles in the development of measures — abstraction, information hiding, and modularity.

2.3 Views

A *design view* is defined in the *IEEE Recommended Practice for Software Design Descriptions* as "a subset of design entity attribute information that is specifically suited to the needs of a software project activity." (IEEE Standard 1016-1987). By focusing on a specific aspect of a design, a view helps to focus attention on relevant issues. A view will normally be cross-sectional: it will include the same kind of information about a number of entities within a design. The following views are used in our framework:

- Code — Complete text of the code or PDL available for each compilation unit of the design
- Compilation Unit Dependency — Identifies direct dependency relationships between compilation units

- Compilation Units — Identifies compilation units in the design
- Imported Name Reference — Identifies imported names that are actually referenced
- Intra-Compilation Unit Name Reference — Provides information about references to names declared in a compilation unit from within that same unit
- Name Importation — Identifies a three-way relationship between a compilation unit, a name defined in that compilation unit, and a second compilation unit that "imports" the name
- Name Space — Identifies every distinct name within the design
- Packages — Identifies packages within the design and gives high-level information about each package
- Subprograms — Identifies all subprograms at any level within the design
- Tasks — Identifies all tasks at any level in the design
- Type Dependency — Identifies a relationship between two types, one of which depends on the other in its definition
- Structure Chart — Provides the subprogram call graph of a program or task

Table 1

Mapping Between Quality Factors and Software Engineering Principles

Reference Quality Factors \ Reference Principles	Software Engineering									
	Abstraction	Completeness	Confirmability	Explicit Interfaces	Information Hiding	Linguistic Modular Units	Minimal Interfaces	Modularity	Open-Closed	Weak Coupling
Correctness		R/A	R	M	M	M	M		M	M
Efficiency *										
Expandability					M	M	M	A	M	M
Flexibility	R	R	R		R			R/A		
Integrity *										
Interoperability				M	M	M	M	A	M	M
Maintainability					A			A		
Portability								A		
Reliability	R				R					
Reusability	B			M	M	M	M	A	M	M
Safety *										
Survivability								A		
Usability	R				R					
Verifiability					A			A		

R - Ross A - Bowen
M - Meyer
B - Booch

* Not currently mapped

2.4 Measures

Software design measures characterize some aspect of a design (i.e., some view) in a way that provides a standard of comparison, estimation, or judgment. As discussed above, we wish to use measures that are at least intuitively related to software engineering principles and, through the principles, to software quality factors. Usually, however, a measure will fall short of capturing the full breadth of a software engineering principle. We may intuit that a measure is indicative of a principle, but will rarely agree that the measure *defines* the principle. The relationship between measures and principles is many-to-many: many measures may be related to one principle, and many principles may be tied to a single measure.

Measures may be either objective or subjective. An automated tool may be available for the computation of objective measures. Subjective measures typically are used to indicate the presence or absence of a feature or to characterize some characteristic along a scale from "poor" to "excellent." While subjective measures require human judgment, they should have well defined measurement procedures such as specific guidelines.

Table 2 lists the measures currently included in the design assessment framework.

Table 2
Design Assessment Measures

Measure	Scale	Views	Principles
Visible Declarations in a Library Package	Ratio	Name Space	Abstraction, Information Hiding, Modularity, Weak Coupling
Imported Declarations for a Compilation Unit	Ratio	Name Space, Name Importation	Minimal Interfaces
Imported Declaration Utilization by Compilation Unit	Ratio	Compilation Units, Imported Name Reference	Information Hiding
Recompilation Effect	Ratio	Compilation Unit Dependency	Minimal Interfaces, Modularity
Package Distribution	Nominal, Ratio	Compilation Unit, Name Space	Abstraction, Explicit Interfaces, Information Hiding, Weak Coupling, Linguistic Modular Units

Table 2
Design Assessment Measures (continued)

Measure	Scale	Views	Principles
Abstract Data Type (ADT) Classification	Nominal	Compilation Unit, Name Space	Abstraction (Data), Information Hiding, Linguistic Modular Units, Modularity, Open-Closed, Weak Coupling
ADT Completeness	Ordinal	Compilation Unit, Subprograms	Abstraction (Data), Completeness
User-Defined Exception Ratio	Nominal, Ratio	Compilation Unit, Name Space	Abstraction (Data), Completeness
ADT Exported Types vs. Total Exported Type Ratio	Ratio	Compilation Unit, Name Space	Abstraction (Data), Information Hiding
Subprogram Parameter Count	Ratio	Compilation Units, Subprograms	Explicit Interfaces, Modularity, Weak Coupling
Composite Type Parameter Usage	Ratio	Subprograms, Name Space, Intra-compilation unit Reference	Weak Coupling
Fraction of Ada Types	Ratio	Compilation Units, Name Space, Type Dependency	Abstraction, Information Hiding
Fraction of Objects of a Type	Ratio	Name Space, Type Dependency	Information Hiding
Data Complexity	Ratio	Type Dependency	Abstraction (Data), Information Hiding
Decision vs. Actual Processing Distribution	Nominal	Code	Modularity
Fan Out	Ratio	Structure Chart	Confirmability, Modularity, Uniformity

Table 2
Design Assessment Measures (concluded)

Measure	Scale	Views	Principles
Average Layered Virtual Machine (LVM) Depth	Ratio	Structure Chart	Confirmability, Modularity, Uniformity
Subprogram References	Ratio	Imported Name References	Confirmability
Efferent/Afferent Systems	Nominal	Code	Modularity
Concurrency Structure	Nominal	Tasks	Confirmability, Linguistic Modular Units, Uniformity
Task Classification	Nominal	Tasks, Code	Abstraction, Information Hiding, Modularity, Open-Closed, Weak Coupling

2.5 Design Representations

A design representation is a structured representation of a view. A design representation has a well defined syntax and semantics. We are most interested in representations that are stored electronically and are processable by automated tools. Design representations may be provided by the software designer (e.g., data-flow diagrams, PDL, source code), or may be derived from other design representations by tools (e.g., structure charts).

Examples of design representations include:

- Source code
- Booch diagrams (Note the distinction: there is a *view* of inter-unit dependency that is independent of any representation, while a Booch diagram is a specific representation of this view.)
- Buhr diagrams (An alternative representation that contains much of the same design information, i.e., corresponds to roughly the same view, as a Booch diagram.)
- Data-item cross references stored in a database

Various design representations corresponding to the same view may be used for particular purposes. For example, a graphical representation of certain information may be suited to a human user, while a textual form may be better suited for input into an automated tool. Tools normally require specific design representations, which can lead to difficulties in attempting to integrate tools from various sources into an integrated tool suite.

2.6 Tool Support

As has been suggested above, automated tools are essential to the assessment process. We can identify these broad categories of assessment tools:

- Characterization tools assist the assessor in describing the software project that produced the design under assessment, as well as high-level characteristics of the software itself. This information can be stored in a database to facilitate comparisons across projects or within a project over time, and to support empirical research.
- Browsing tools assist the assessor in understanding the design. They may also provide the capability of answering ad hoc queries about a design. (e.g., "Where is variable X defined? Where is it used?") The EASE tool is an example of such a tool.
- Design representation tools convert one representation to another, or generate or derive one representation from another (e.g., a tool to produce a structure chart from source code).
- Metrics tools compute measures.
- View presentation tools extract the needed information from a design and present it to the assessor in a particular form such as a histogram or cumulative distribution curve.

3. Formulating an Assessment Strategy

Each project under assessment will have unique characteristics that influence the type of assessment performed. For instance, the application being built may be a real-time avionics system in which case reliability is a key concern. Such characteristics help the assessor understand the context of the assessment and serve to focus the assessment. A set of project profiles is completed by the assessor as a mechanism for capturing project-specific characteristics. The information captured in these profiles, in conjunction with the assessment framework, is used to develop a project-specific assessment strategy as illustrated in figure 2.

3.1 Characterizing the Project

Our methodology utilizes five project characterization profiles to establish the assessment context. Each profile defines a set of project variables related to its problem domain, acquisition process and/or development process. For instance, "design method" is a project variable that characterizes an aspect of the development process. The profiles presented here are synthesized from other similar assessment/analysis vehicles (Shultz, 1988; Humphrey, 1987) and are based on a software development effort under DOD-STD-2167A. These profiles can be adapted as necessary. Each of the characterization profiles is summarized here:

- Project Profile — The project profile captures characteristics of the problem domain, the acquisition process and the development process at a high level. It provides an overview of the project in terms of its government sponsor, application area, type of development, required quality factors, life cycle model, and class of system being built.
- Architecture Profile — The architecture profile provides a high level description of the operational system in terms of its hardware and software. It characterizes the

system configuration, the level of custom hardware, the Computer Software Configuration Item (CSCI)-level architecture, and the allocation of software to hardware.

- CSCI Profile — The CSCI profile captures the most detailed information about the problem domain and the development process. It has several sub-characterizations: general CSCI, software construction, developer, and development environment.
- Developer Profile — The developer profile identifies the respective responsibility of both prime contractors as well as any subcontractors. It also identifies the developer's past experience with this type of application.
- Assessment Profile — The assessment profile provides information pertaining to the assessment activity itself. It identifies the project assessed, the sponsoring organization supported, and the life cycle point at which the assessment was performed.

3.2 Navigating through the Assessment Framework

The conceptual framework for design assessment identified the following concepts: quality factors, principles, views and view classes, measures, and design representations. The following sections address how an assessor would navigate through this framework, using the project characterization data to narrow the assessment focus.

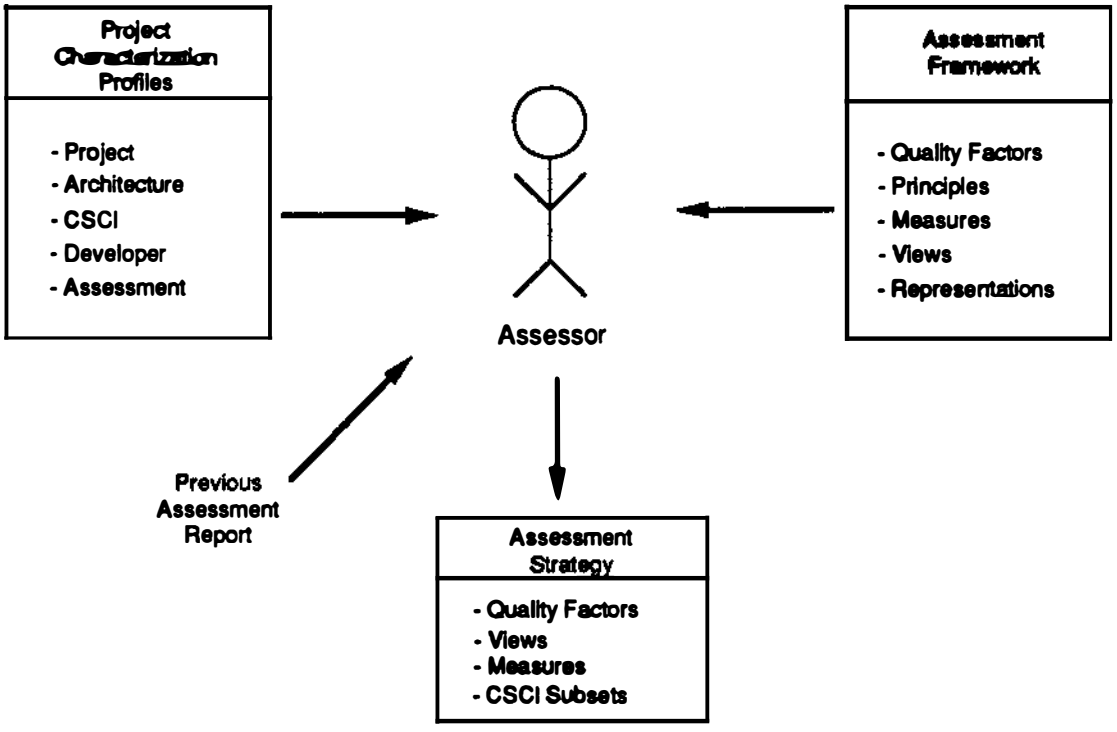
Navigation through the framework occurs from both end points: quality factors and design representations. Quality factors are used to identify principles and representations are used to determine available views. Both principles and views map to the measures that will characterize some aspect of the design. (Refer to figure 1.)

A given design product will use various design representations such as Ada PDL, Booch diagrams, Finite State Machines (FSMs), etc. Depending on the representations used, certain views can be extracted or derived. The identification of available views is important because the derivation of each measure requires specific inputs, and these inputs are obtained from the representations. Some representations may provide insufficient information to extract or derive the necessary view. In the current framework the only representation supported is Ada PDL and/or code. All defined views are based on this representation.

Given the relevant quality factors and available representations, the assessor can now determine which views must be generated and which measures to apply. An assessment could proceed at this point. However, further refinements might be desired. Refinements are based on data collected in the project profiles. The following guidelines provide insight into how the selection of possible elements in the assessment framework can be made and how to extend what is currently available based on the project profiles.

- Quality Factor Selection — if a project has not identified important quality factors the project profiles can be used to target appropriate ones. For instance, if the design incorporates both a significant number of Commerical Off-the-Shelf (COTS) products and reusable components, there may be a need to assess the design for efficiency or interoperability. Similar types of relationships can be drawn from other profile data.

Figure 2
Assessment Strategy Formulation



- View Selection — Available design representations determine what information is available for analysis, and hence which views can be used for an assessment. Project and CSCI profiles can be used to restrict the views that are actually used. For instance, if a particular CSCI has stringent performance requirements, then the focus of the assessment for that CSCI should be primarily the views contained in the performance view class.
- CSCI Selection — The CSCI profile can be used to identify the CSCIs that should receive individual assessments. If a certain CSCI has a number of risks associated with it, then it probably should receive an assessment. For instance, if it has real-time performance constraints, significantly complex interfaces, and volatile requirements then it might be a strong candidate for assessment. The specific quality factors of interest may be reliability and flexibility. Using information from several profiles can assist the assessor in the selection and prioritization of CSCIs for assessment.
- Measure Selection — Available views and relevant principles identify measures that can be potentially applied to the design. Once a set of measures is identified, the detailed information associated with each measure can influence the decision:
  - Is the measure used for detailed analysis or for profiling? Does it provide specific information for a particular type of unit or is the information a cross-sectional view across all units?

- Is there tool support? This is important for extracting extensive information across reasonably-sized applications, and for computing measures based on the information or for generating a graphical representation of the results.
- Is it easily derived from the available design information?
- Is it necessary to derive several intermediate computations or views in order to apply the measure?

- Multiple Assessments — Assessments performed several times over the course of development would allow for a relative, rather than absolute, assessment. By capturing several "snapshots" of the quality of the evolving design, trends might be observed that help to target the assessment. These trends should be evidenced both in the project characterization data and in the assessment results. Information from the previous assessment(s) could be used as additional input to the strategy formulation step for the current assessment.
- Framework Extensions — Given the generality of the framework, expansion and adaptability are possible. If the project characterization process identifies the need for assessment in areas not addressed by the current framework, the assessor could choose to extend it. For instance, if safety is an important quality factor, the principles defined in (Smith, 1988) could be used to develop appropriate measures. Such effort would of course require additional resources beyond those required for a normal assessment. Any extensions developed during an assessment would then be fed back into the assessment framework for future assessors.

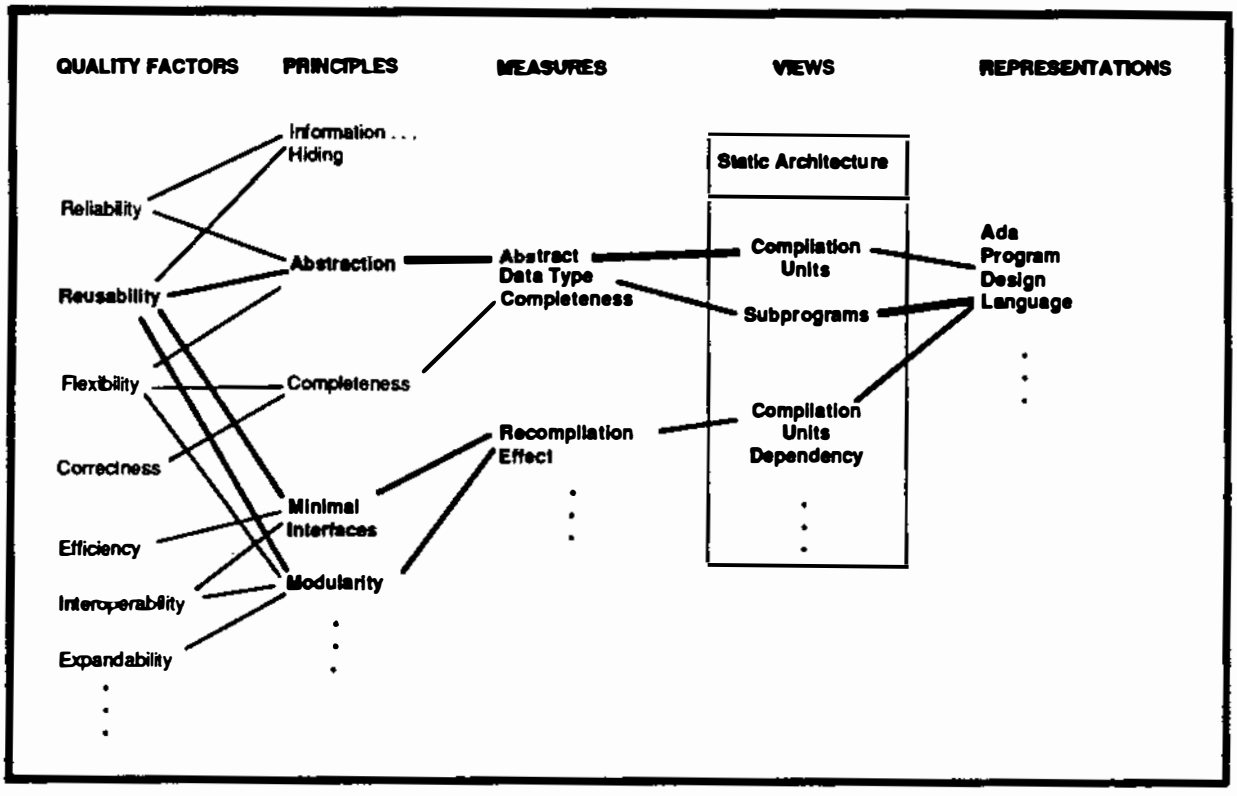
4. Application of the Framework

Use of the framework is illustrated here through a small example. It shows how an assessor identifies measures to be applied and how that assessor interprets the results obtained. This example assumes that a project profile has been performed and that a single quality factor, reusability, is the only relevant one for the design under assessment. Figure 3 illustrates how the identification of relevant quality factors determines the appropriate measures to be applied, given an Ada PDL as the design representation. In this example reusability maps to four principles: information hiding, abstraction, minimal interfaces, and modularity. Two measures — Abstract data type (ADT) completeness and Recompilation Effect — related to these principles are described in detail in the following sections.

4.1 ADT Completeness

A data abstraction can be evaluated for completeness by examining both its operation set and its error handling provisions. This measure examines the visible operations. It determines the completeness of the component by examining the kinds of operations made available to clients. A limited set of operations may restrict the usability of the abstraction in its original context as well as its reusability in other contexts.

Figure 8
Strategy Formulation Example



Profile Derived

A data abstraction will provide operations that can be classified into one of the following four categories (Liskov and Guttag, 1986). To classify the visible operations of an Ada package, the characterizing signatures given by Nielsen and Shumate (1988) are used, with some refinements.

- Primitive Constructors — used to create instances (objects) of the declared data type. In most cases there will not be primitive constructors in an Ada abstraction since objects of the visible type are simply declared as needed in the client. A constant can be considered a primitive constructor since it declares an object of a given type that has a non-mutable value (Embley and Woodfield, 1988).
- Constructors — given an object(s) of the type as input, it creates another object of that type (i.e., integer multiplication). A procedure that has formal parameters of both "in" and "out" for the package's exported type is a constructor. A function

that serves as a constructor returns an object of the visible type as the result and receives parameters of that type as input.

- Mutators — used to modify an object of the given type. A procedure that has the visible type as an "in out" parameter is a mutator.
- Observers — provide information about an object of the given type. It uses object(s) of the type as input and returns the results of another type (i.e., an "is_empty" function on a stack type that returns a boolean result). A procedure that has the visible type as an "in" parameter and another type as an "out" parameter or a function that has the visible type as an "in" parameter and another type as the result type are classified as observers.

Note that procedures that make use of access types as formal parameters hide the true operation being performed. An operation that has a parameter of access type with mode "in" may be a mutator since it designates an object which can be accessed and modified by the operation. (Recall that an object designated by an access type is a "variable"). Therefore, when access types are used in the formal parameters of abstractions, the type of operation cannot be determined without examining the implementation.

This profile uses as input the package specifications classified as closed or open ADTs (Nielsen and Shumate, 1988). A completeness profile is derived for each package as illustrated in figure 4. It is used to classify each visible operation in one of the four categories described above.

Figure 4
ADT Completeness Profile

Package	Visible Type	Visible Operation	Mode Profile				Operation* Signature	Adequate
			In	In Out	Out	Result		
P_1	T_1	O_1	–	–	x	–	PC	NO
		O_2	x	–	–	–	O	
P_2	T_2	O	–	x	–	–	M	NO
		O_1	–	x	–	–	M	
		O_2	–	x	–	–	M	
		O_3	x	–	–	–	O	
		O_4						

* Possible options are:
PC = Primitive Constructor
C = Constructor
M = Mutator
O = Observer

Definition of Measure

According to Liskov and Guttag (1986), an adequate abstraction provides one of two possible combinations of operations:

- Primitive Constructor, Constructor, and Observer
- Primitive Constructor, Mutator, and Observer

This measure determines whether a given ADT is adequate. If an ADT package provides one of the above combinations of operations it is classified as adequate. In reality, multiple mutators, constructors and observers may be necessary to provide a complete abstraction. (See Booch [1987] for some thoughts.)

Interpretation of Results

If it is desirable to create reusable components within the application, the degree of completeness can be used as one gauge for determining which components are potentially reusable based on the operation set provided to a client. Completeness of operations also minimizes the need for clients to circumvent the operations provided, especially in the use of open ADTs where structures can be manipulated outside the abstraction's operation set.

4.2 Recompilation Effect

Ada support for separate compilation introduces direct and indirect dependencies among units. The measures presented here examine recompilation effect, based on the types of compilation dependency that can be introduced in an Ada application. For large applications, the recompilation effect of changes to a unit may be significant. Large recompilation effects are indicative of possible poor cohesion in the associated units. They may also lead to excess consumption of development or support system resources.

A number of compilation dependencies are defined by the Ada Lanuage Reference Manual (ANSI/MIL-STD-1815A, 1983). For example, a change made to a package specification requires that the corresponding package body be recompiled; a change in the specification of a library unit that appears in a context clause requires recompilation of the dependent unit.

Profiles Derived

Determining the total recompilation effect for a compilation unit requires that a trace be made through both its direct and indirect dependencies. In the example shown in figure 5, CU1 is indirectly dependent on CU6 since CU2 is the specification and CU6 is the body of a program unit. Therefore, a change to CU1 forces a recompilation of CU2 that *indirectly* forces a recompilation of the body contained in CU6. Therefore, the total recompilation effect of a change to CU1 is recompilation of CU2, CU3, and CU6.

Definition of Measure

The average recompilation effect for any compilation unit can be computed as follows:
ARCE = Σ_i RCE(i) / NCU where,
RCE(i) = the total number of compilation units that must be recompiled given a change to compilation unit i, and
NCU = the total number of compilation units.

Interpretation of Results

A high average recompilation effect suggests that changes may induce significant recompilations during implementation and maintenance. This can lead to excess consumption of resources (i.e., computer time). It may also lead to schedule problems, since it may be impossible to achieve quick turn-around to correct errors discovered during testing. This measure can only be considered suggestive, however, since not all compilation units are equally likely to be changed. Specifications will tend to have a larger recompilation effect than bodies. However, in a well designed system, bodies should be more likely to change than specifications.

Figure 5
Recompilation Effect Profile

Depends on: / Compilation Unit	CU_1	CU_2	CU_3	CU_4	CU_5	CU_6	. . .	CU_n	Recompilation Count
CU_1		X	X						3
CU_2						X			1
CU_3									
CU_4					.				.
CU_5					.				.
CU_6					.				.
.									
.									
.									
CU_n									

Individual units with a high recompilation effect should be examined to see if they exhibit poor cohesion. If such units are judged to be reasonably cohesive, consideration should be given to minimizing the likelihood that they will be changed. In some cases this can be accomplished by designing packages so that changes are more likely in the package body than in the specification. For example, minimizing change can be accomplished by placing the initialization of object values in the package body rather than in the specification.

5. Assessor Workbench

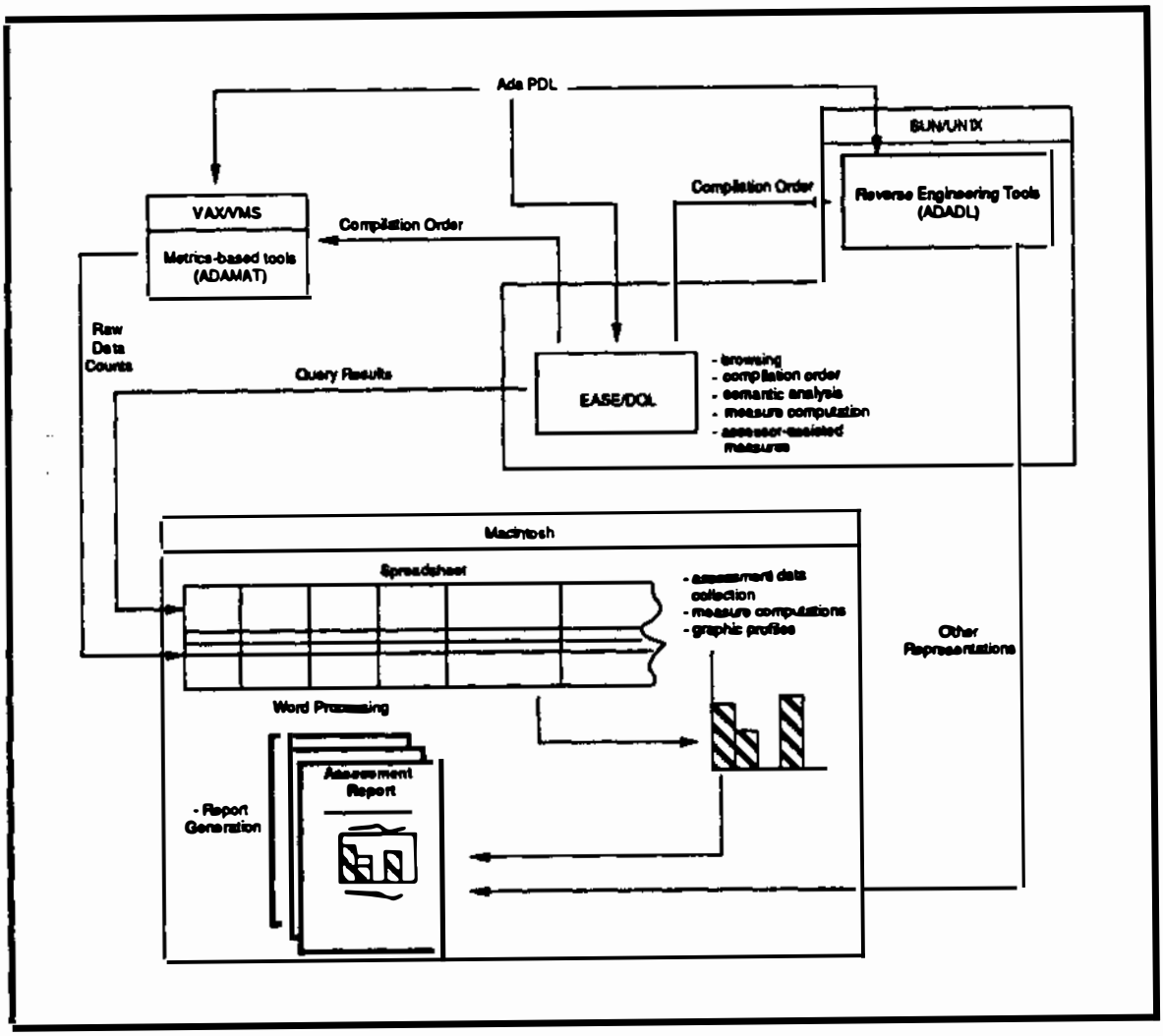
In the context of the present framework, tools are used to support project characterization, to allow an assessor to browse through available design data, to extract data corresponding to a particular view from one or more design representations, to convert one

design representation to another, and to compute measures from design representations. Tools are essential to the effective application of our methodology because of the volume of design materials that usually must be reviewed in the course of an assessment.

Existing tools are being integrated into an assessor's workbench as illustrated in figure 6. The basic capabilities of these tools are being adapted to support the established assessment framework. In particular, the following work is ongoing:

- Metrics tools are being used as a source of raw data counts for computation of the framework-defined measures. Although these tools provide their own unique set of metrics, the raw data serve as the input for this work.

Figure 6
Assessor's Workbench: Initial Concepts



The AdaMAT tool by Dynamics Research Corporation (DRC) is the principal metric tool currently being used. It computes a wide collection of metrics for individual units. Underlying each of the DRC-defined metrics is a collection of primitive data counts used to compute each metric. It is these primitive data counts that are of interest for design assessment work. These primitives serve as the input for computation of a subset of the measures defined in the framework.

The information obtained from AdaMAT is limited to intra-unit analysis. Inter-unit data for Ada units requires knowledge of context dependencies and scoping rules of the language. Although this support is not currently available in AdaMAT, an in-house tool provides a partial capability, as described below.

- The Electronic System Division Acquisition Support Environment (EASE) is a Sun/UNIX based toolset. Of particular interest to the design assessment work is the query language capability within EASE. The Descriptive Intermediate Attributed Notation for Ada (DIANA) (Evans, 1983) Query Language (DQL) (Byrnes, 1989) is a set of primitive search and combination operations for querying the DIANA intermediate form of Ada source code, much like a conventional database can be queried. DQL can be used by an assessor to determine how well the Ada software conforms to design standards, to compute metrics based on the software's structure, and to browse through the software. DQL can also be used as an integration layer on which new and existing tools may be implemented to extract relevant information about Ada source code.
- The spreadsheet/macro facility of the Macintosh/EXCEL product is being used to generate some of the identified measures. This product will also provide graphics support for the generation of design profiles based on the computed measures.
- Reverse engineering tools will be used to derive alternate representations of the design such as call trees and dependency hierarchies for initial understanding and validation of the design.

The assessor's workbench is an evolving concept. The initial approach capitalizes on existing technology by integration and customization. Continued work on this tool support is necessary to produce the full support needed for assessment of large-scale systems.

6. Future Directions

The design assessment methodology described here is by no means complete. We recommend that future work in this area concentrate on the following (in approximate order of priority):

- The overriding need is to validate the described framework. Although our methodology largely builds on previously published work, it employs this work in a process that should be validated on actual assessment projects. Ideally, validation would begin with appropriately scaled pilot projects.
- Adequate tool support for the methodology must be developed. The data generated by this methodology can quickly overwhelm an assessor. While some work has been done to identify appropriate tools and provide necessary extensions and utilities, additional work is needed. Most of the progress thus far has been oriented toward

analyzers that can be applied to PDL. Some associated utilities have also been developed, which can generate views, profiles, and measures from analyzer outputs. However, it is unreasonable to assume that design products will always be delivered in the form of PDL. If the designs are expressed in terms of the outputs of a Computer-Aided Software Engineering (CASE) product, then the necessary infrastructure must be developed to construct views, profiles, and measures from these outputs.

- The design assessment framework may benefit from a number of extensions. Work should proceed to identify additional measures to characterize the design, and to validate these measures. Linkages between adherence to principles and ultimate software quality were assumed, since very little empirical study has been done in this area, especially for Ada-oriented designs. Additional measures are especially needed in the areas of concurrenc and performance assessment. The present effort could also logically evolve into application-domain-specific evaluations.
- The technology and perspectives embodied in the design assessment methodology should be disseminated for use among a wider audience. Likewise, where others have pursued successful research in this area their results should be integrated as appropriate. For example, we are currently investigating work performed at the Virginia Polytechnic Institute and State University (VPI) through funding by the U.S. Navy's Systems Research Center (Arthur and Nance, 1987). Their approach is similar in many respects and they have performed some case studies to begin to validate their approach. The VPI research is additionally pursuing complementary directions such as design methodology evaluations and documentation analysis.

A substantial effort is required to bring more formal design assessment methodologies and supporting technology into practice in the software engineering community. Initial efforts seem promising and benefits are expected with even interim products. Others are encouraged to collaborate in advancing the state of the practice in design assessment.

References

Arthur, James D. and Richard E. Nance. April, 1987. "Developing an Automated Procedure For Evaluating Software Development Methodologies and Associated Products." Technical Report SRC-87-007. Blacksburg, VA: Systems Research Center and Department of Computer Science, Virginia Polytechnic Institute and State University.

Boehm, Barry W. 1981. *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, Inc.

Booch, Grady. 1987. *Software Engineering with Ada,* Second Edition. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc.

Bowen et al. February, 1985. "Specification of Software Quality Attributes: Software Quality Specification Guidebook." Volume II (of III). RADC-TR-85-37. Rome, NY: Rome Air Development Center.

Byrnes, Christopher. 1989. "A DIANA Query Language for the Analysis of Ada Software." MTP-281. Bedford, MA: The MITRE Corporation.

Byrnes, Christopher. February, 1988. "ESD Acquisition Support Environment (EASE)." MTR-10371. Bedford, MA: The MITRE Corporation.

Curtis, Bill. September, 1980. "Measurement and Experimentation in Software Engineering." *Proceedings of the IEEE.* Vol. 68, no.9, pp. 1144-1157. IEEE Computer Society.

Department of Defense. February, 1988. DOD-STD-2167A. Washington, DC: Department of Defense.

Deutsch, Michael and Ronald Willis. 1988. *Software Quality Engineering: A Total Technical and Management Approach.* Englewood Cliffs, NJ: Prentice-Hall, Inc.

Embley, David W. and Scott N. Woodfield. April 11-15, 1988. "Assessing the Quality of Abstract Data Types Written in Ada," *Proceedings of 10th International Conference on Software Engineering.* pp.144-153. Singapore, China.

Evans, A. and K. J. Butler. 1983. "Descriptive Intermediate Attributed Notation for Ada Reference Manual." TL-83-4. Pittsburgh, PA: Tartan Labs.

Grady, Robert and Deborah Caswell. 1986. *Software Metrics: Establishing a Company Wide Program.* Englewood Cliffs, NJ: Prentice Hall.

Humphrey, W.S. and W.L. Sweet. September, 1987. "A Method for Assessing the Software Engineering Capability of Contractors." CMU/SEI-87-TR-23 (ESD/TR-87-186) Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

IEEE. August, 1983. "IEEE Standard Glossary of Software Engineering Terminology." ANSI/IEEE-Std-729-1983. New York, NY: The Institute of Electrical and Electronics Engineers, Inc.

IEEE. October, 1987. "IEEE Recommended Practice for Software Design Descriptions." IEEE-Std-1016-1987. New York, NY: The Institute of Electrical and Electronics Engineers, Inc.

Liskov and Guttag. 1986. *Abstraction and Specification in Program Development.* Cambridge, MA: The MIT Press.

Meyer, Bertrand. 1988. *Object-oriented Software Construction.* Great Britain: Prentice Hall International (UK) Ltd, 66 Wood Lane End, Hemel Hempstead, Hertfordshire, HP2 4RG.

Nielsen and Shumate. 1988. *Designing Large Real-Time Systems with Ada.* New York, NY: McGraw-Hill.

Ross et al. May, 1975. "Software Engineering: Process, Principles, and Goals." *IEEE Computer*, pp. 17-27. IEEE Computer Society.

Schultz, Herman P. May, 1988. "Software Management Metrics." M-88-1. Bedford, MA: The MITRE Corporation.

Slonaker, P., et al. March, 1987. "Development of Multi-Tasking Software in Ada — A Case Study," *Proceedings of the Fifth National Conference on Ada Technology*, pp. 304-317.

DOUGLAS W. COOKE

Mr. Cooke is a member of the technical staff in the MITRE Corporation's Washington C^3I Software Technology Center. His interests are in software engineering methodologies, design assessment, simulation, and Ada real-time design. Mr. Cooke holds a M.A. in Mathematics from the Univesity of Colorado and a B.A. from San Jose State College. He is a member of the ACM.

WILLIAM EVANCO

William Evanco received his B.S. in physics from Carnegie-Mellon University and his Ph.D. in theoretical physics from Cornell University. His research interests include the statistical analysis of software quality in terms of design measures, and software performance analysis using discrete event simulation models. He is currently employed in the MITRE Corporation's Washington C^3I Software Technology Center.

ROBERTA HUTCHISON

Roberta Hutchison is an Associate Department Head in the MITRE Corporation's Washington C^3I Software Technology Center. In addition, Ms. Hutchison currently assists the MITRE Washington C^3I Division's Chief Engineer in overseeing the Division's other Technical Centers.

FRANCIS X. MAGINNIS

Mr. Maginnis is a Principal Engineer in the MITRE Corporation's Washington C^3I Software Technology Center. His interests include software-engineering methodologies, parallel and distributed processing, simulation, and Ada-oriented design and programming. Mr. Maginnis holds a M.S. degree in Computer Science from Stanford University and a B.A. in Mathematics from the College of the Holy Cross. He is a member of ACM, the IEEE Computer Society, American Association for Artificial Intelligence, and the Society for Computer Simulation.

DIANE E. MULARZ

Ms. Mularz received the B.S. degree in Mathematics from Indiana University of Pennsylvania and the M.S. degree in Computer Science from Johns Hopkins University. She is currently employed as a Lead Scientist in the MITRE Corporation's Washington C^3I Software Technology Center. Her current interests include software engineering environments and software quality assessment technology. Ms. Mularz is a member of the ACM and the IEEE Computer Society.

1990 PROCEEDINGS ORDER FORM

Pacific Northwest Software Quality Conference

To order a copy of the 1990 Proceedings, please send a check in the amount of $30.00 to:

PNSQC
P.O. Box 970
Beaverton, OR 97075

Name _____

Title _____

Affiliation _____

Mailing Address _____

City, State _____ Zip _____

Daytime Telephone _____