View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NASA Technical Reports Server

Software: Where We Are & What is Required in the Future

Jerry Cohen

Boeing Aerospace and Electronics

High
Technology
Center

Flight Critical Software: Current Status and Future Direction

Gerald C. Cohen
Boeing Aerospace & Electronics
High Technology Center

October 31, 1990 8:34 AM HTC/001

The Programmers
ENVIRONMENT

High
Technology
Center
BOEING

- High integrity considerations
- Hard real-time constraints
- Implications of a still evolving systems architecture
- Need to meet delivery schedules with high productivity
- Evolving requirements & specifications

High
Technology
Center

BOEING

RESULTS

CASE 1

High
Technology
Center

BOEING

- Triplex Digital Flight Control System
  - Not synchronized
- Analog backup
- Each computer samples sensors independently, uses averages of good channels

CASE 1

Flight

- Asynchronous operation, skew, and sensor noise led each channel to declare others failed
- Analog backup not selected
- No hardware failures had occurred

CASE 1

Analysis

- Failure traced to roll axis software switch
- Sensor noise and a synchronous operation caused one channel to take a different path through the control lows
- Fix was to vote software switch
- Extensive simulation and testing performed
- Next flight – same problem
  - Although switch value was voted, unvoted value was used

CASE 2

High
Technology
Center

BOEING

- Single failure in redundant uplink hardware
- Software detected this – continued operation
- Would not allow landing gear to be deployed
- Aircraft landed with wheels retracted – sustained little damage
- Traced to timing change in the software that had survived extensive testing

Saab Grippen Flight Test Program

High
Technology
Center

BOEING

- Unstable aircraft
- Triplex DFCS with analog backup
- Yaw oscillations observed on several flights
- Final flight had uncontrollable pitch oscillations
- Crashed on landing
- Traced to control laws

B-1B Defensive Avionics

- fundamental flaw in system architecture

Present Day Problems

High
Technology
Center

BOEING

- Requirements are incomplete
- Specifications are incomplete or inconsistent
- No way of proving specification satisfies requirements
- Implementation performed on host machine
  - No relationship to target machine
  - Different operating systems on both machines
  - No way to guarantee real time operation
- Enormous cost overruns
- Late delivery

- Software delivered does not behave as intended
- Validation and verification
  - practically impossible for large programs
  - state space explosion
- Testing procedures are ad-hoc
- No general architecture
- Different languages for different phases of life cycle
- High maintenance costs

High
Technology
Center

BOEING

It appears that 60-70% of all software problems are related to requirements/specifications not being complete or inconsistent

Present Day Tools

High
Technology
Center

BOEING

Case Tools

- Bubble charts (Yourdon, etc.)
  - Data flow
  - Control flow
  - Bookeeping

They do not:

- perform reliability analysis
- perform architecture design
- perform component design
- perform & produce trade studies
- perform testing
- produce test procedures
- perform configuration management

- They do:
  - Support functional decomposition
  - Interfaces allocated to components
  - Functionality derived from constraints and performance

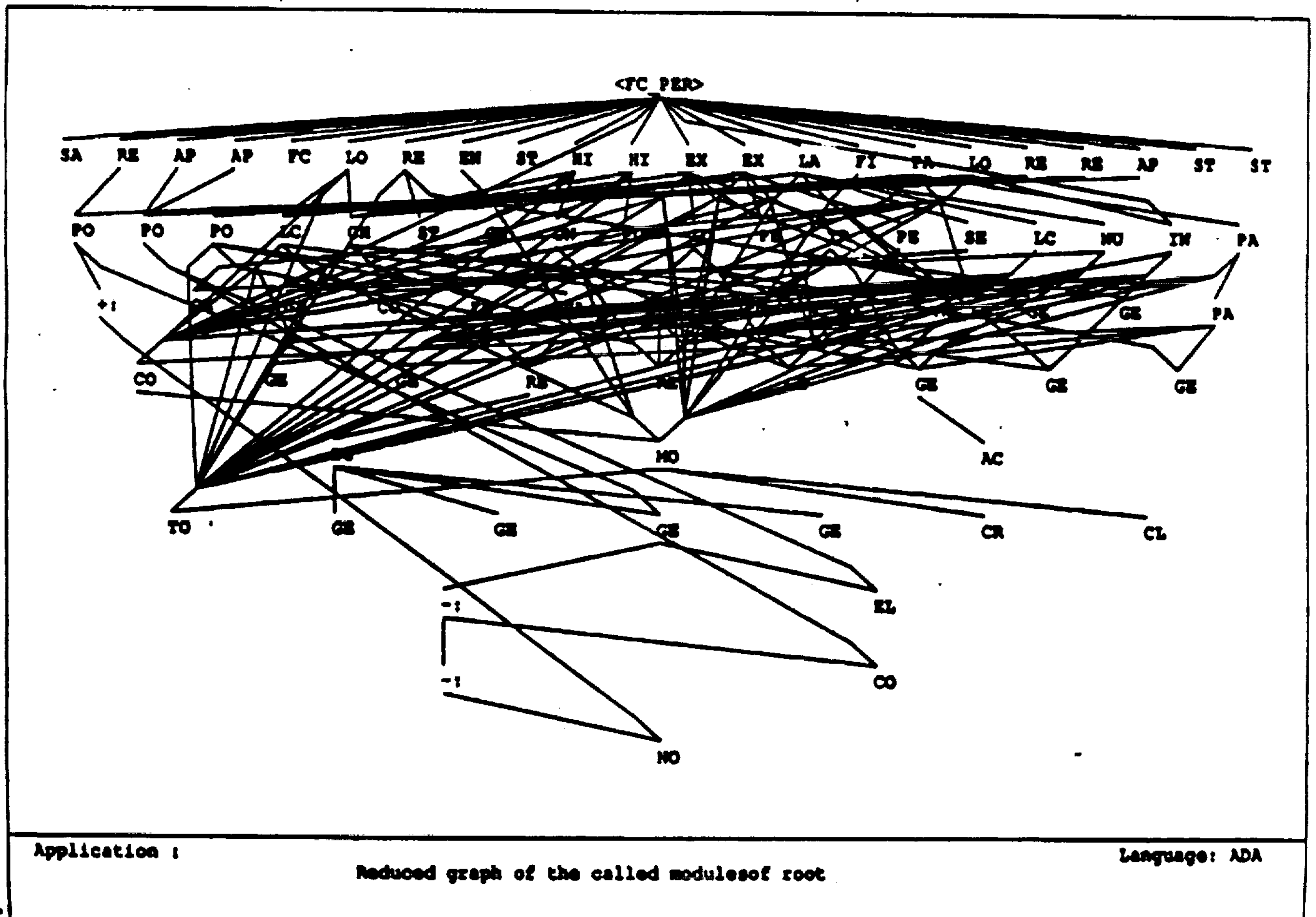
Payoff:

- Interfaces defined between functions
- Behavior is represented by functions
- Constraints influence behavior

Overall Benefits

- Provides integrated requirements database
- Supports impact analysis
- Identifies and reduces risk
- It supposedly adds structure to the requirements/specification phase

Analysis Tools
(reverse engineering)



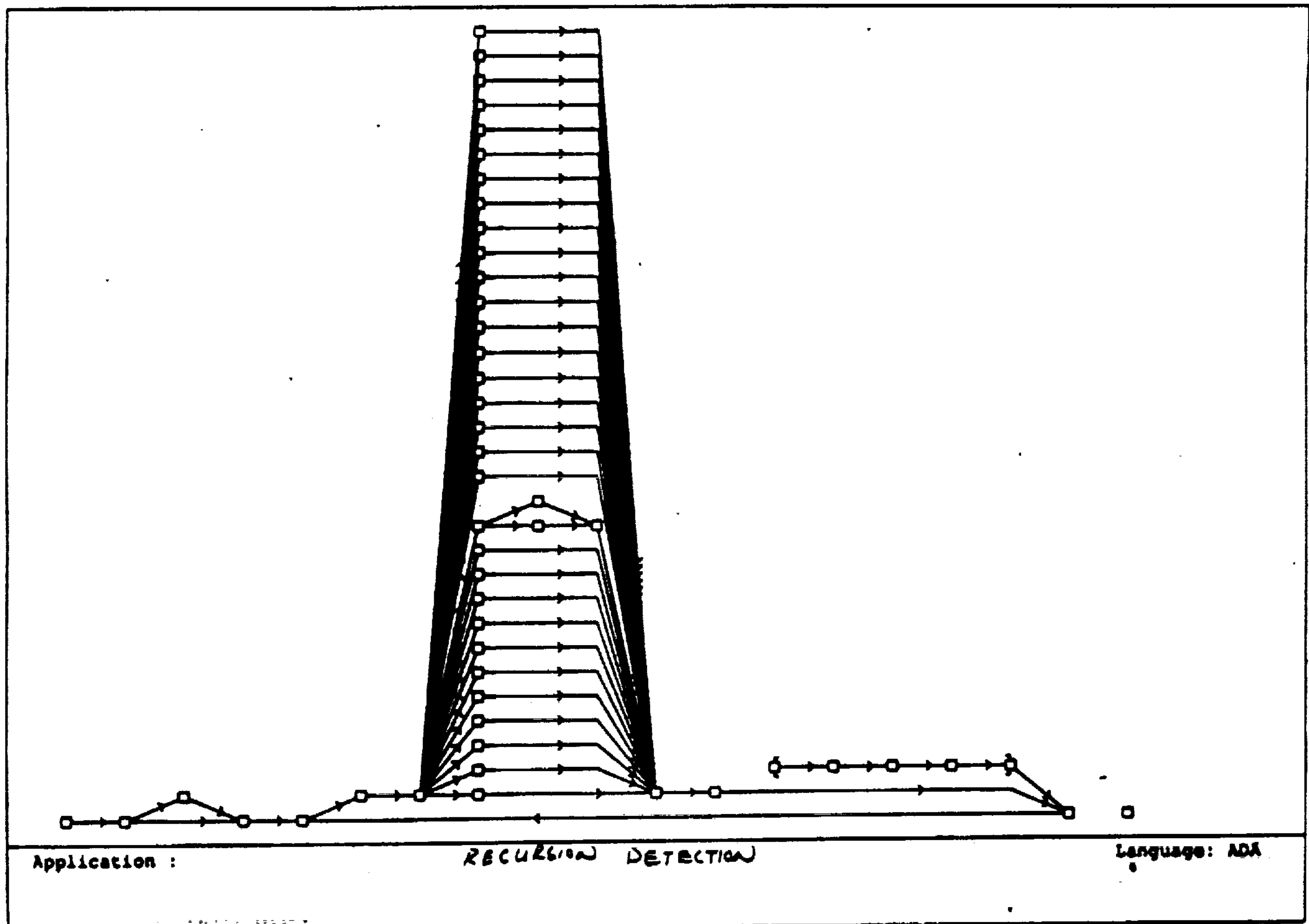
ORIGINAL PAGE IS
OF POOR QUALITY

Calling Tree

- Reuse of modules

  (in general doesn't occur in hardware design for a particular function)

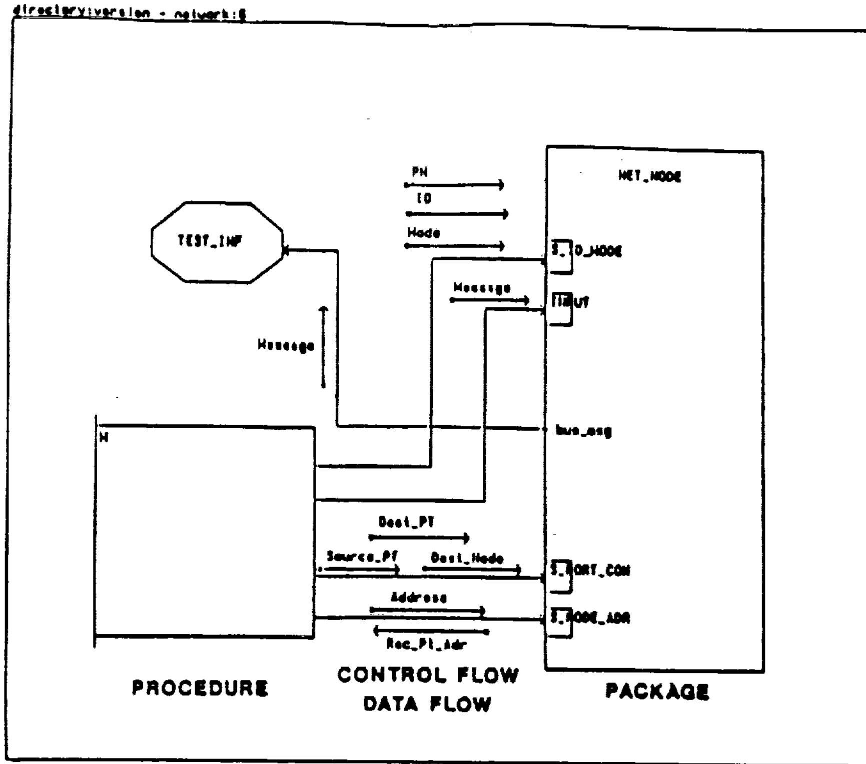
- Shows complexity
- Real time analysis is a problem



Automatic Code Generators

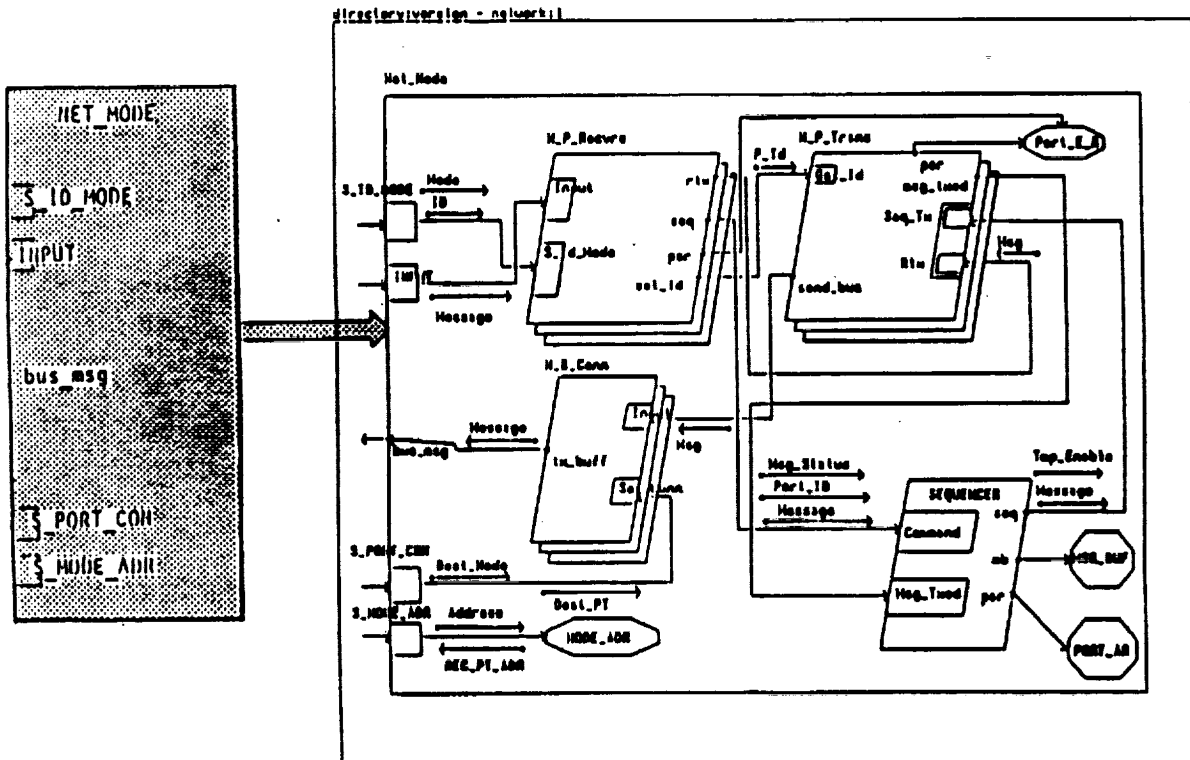
- Caede
- Matrix

REDUNDANT DATA BUS SOFTWARE



71019 M3074-12

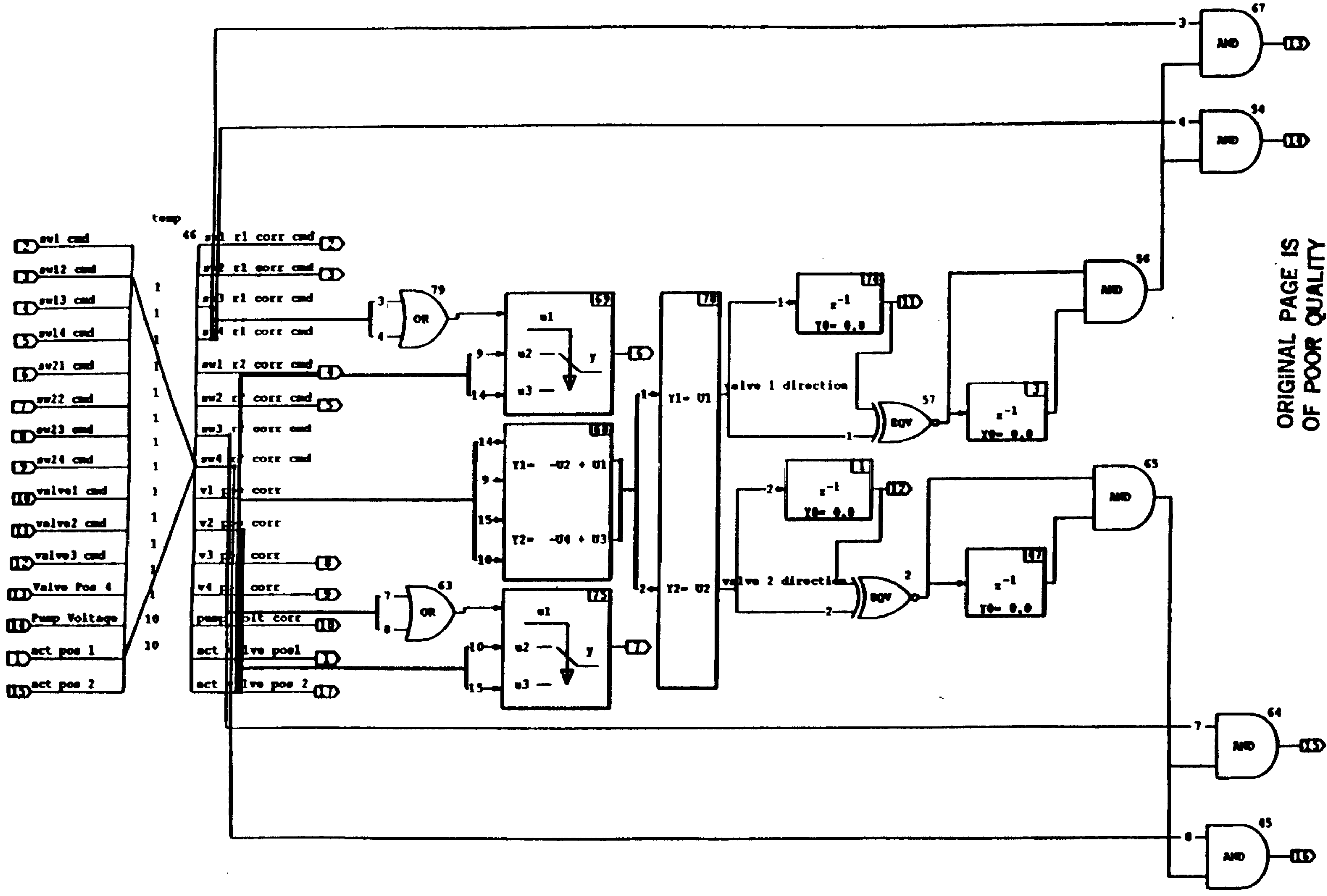
REDUNDANT DATA BUS SOFTWARE



71019 M307

ORIGINAL PAGE IS
OF POOR QUALITY

Sampling Interval	First Sample	Ext.Inputs	Ext.Outputs	Enable
1.	0.	15	17	Parent



MATRIX INPUT

ORIGINAL PAGE IS OF POOR QUALITY

High
Technology
Center

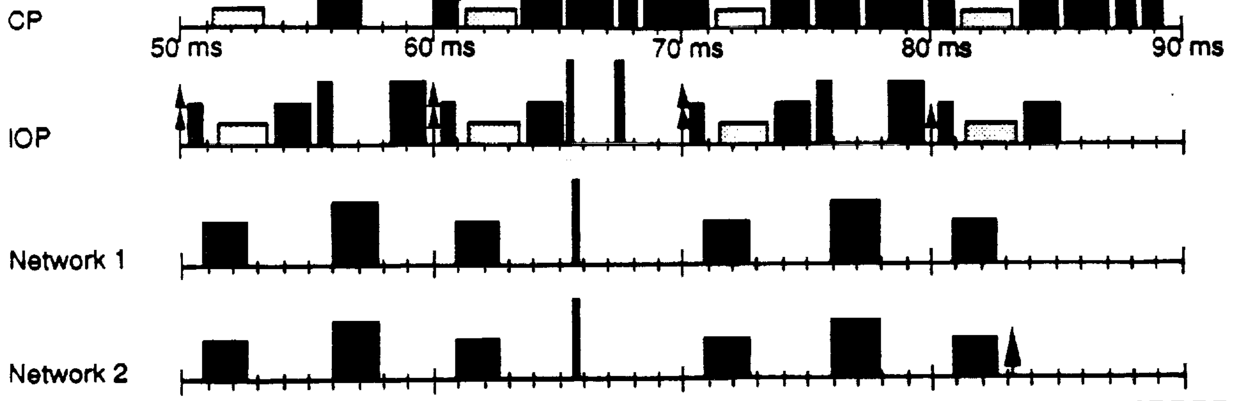
BOEING

Future

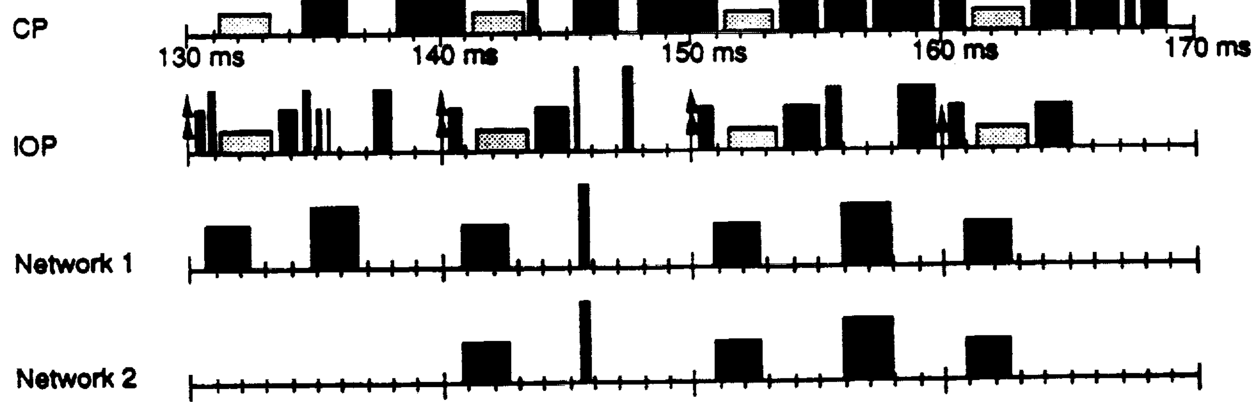
High
Technology
Center

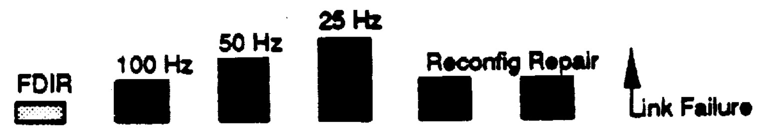
BOEING

- Need systems engineering approach
  - Systems will be more integrated in the future
  - Need better analysis between hardware & software


Frame 1
CP
50 ms
60 ms
70 ms
80 ms
90 ms
IOP
Network 1
Network 2

Frame 2
CP
90 ms
100 ms
110 ms
120 ms
130 ms
IOP
Network 1
Network 2


Frame 3
CP
130 ms
140 ms
150 ms
160 ms
170 ms
IOP
Network 1
Network 2


FDIR
100 Hz
50 Hz
25 Hz
Reconfig Repair
Link Failure

- Need portability
  - Standard interfaces
  - Graphics
  - Data bases

High
Technology
Center

BOEING

- Common software architectures
  - Exist for compilers & operating systems
  - Does not exist for application software (hardware years ahead in this regard)

- Gradual introduction of formal representation for validation & verification
  - Formal representation of requirements and specification

- English Requirements

  Spiral Mode

  a) If unstable, the spiral mode time to double amplitude shall be no less than 20 seconds at speed from 1.2 VS1 to VFC/MFC (Conventional control)

  b) The airplane characteristics shall not exhibit coupled roll-spiral mode in response to the pilot roll commands

  c) Minimum acceptable: the spiral mode time to double amplitude shall be greater than 4 seconds

- **Formal statement of "Spiral Mode" requirements:**

```
a) if Aircraft.State = Unstable then
     if Aircraft.State.Mode = "Spiral" and Aircraft.State.Time = t and
        Aircraft.State.Amplitude = a and
        1.2 * VS1 $<$= aircraft.state.speed $<$= VFC/MFC then
     exists t $<$= t1 $<$= t+20 : Aircraft.State. Amplitude = 2 * a

b) module PilotCommand
     operation RollControl
        postcondition: Aircraft.State.Mode ~= "CoupledRollSpiral"
           . . .
     end RollControl

c) forall s in Aircraft.State :
     if s.Mode = "Spiral" and s.Time = t and s.Amplitude = a
        forall t $<$= t1 $<$= t+4 :
           if s.Time = t1 then s.Amplitude $<$ 2 * a
```


High
Technology
Center

BOEING

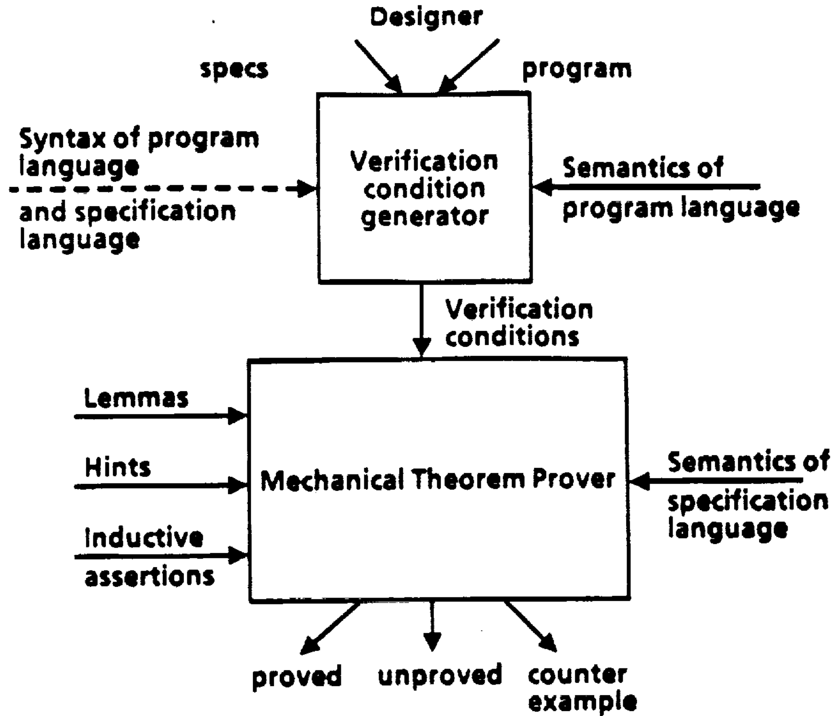
Benefits

- Can prove that specifications satisfy requirements
- Can prove various properties of specifications
  - traceability
  - generate test cases
- Can execute specifications (i.e. OBJ)
  - reasoning about changes

- Need formal verification of software (10-20 years)
  - Actual software
  - Formal proof of automatic code generator

- Need high order language
  - OBJ
    - shorter programs
    - no difference between specification and programming language
    - reuseable code
    - decisions tend to be localized

Detailed View of a Verification System



- Subset of Fortran
- Subset of Pascal
- Subset of Ada
- Subset of "C"
- Gypsy

A Growing Fear

High
Technology
Center
BOEING

"Red Paper"

Bill Totten

President of K.K. Ashisuto

"The Largest Distributor of Independent Software Products in Japan".

"I believe that the United States is in danger of abandoning another vital industry to Japan. This is the computer industry; both computer hardware and computer software.

I see the same pattern of abandonment and surrender now beginning in computers that has occurred before in such industries as motorcycles, automobiles, consumer electronics, office equipment and semiconductors."

"Japan's electronics industry is the worlds best and largest because it is the most competitive. <u>It is competitive because it is based on standards rather than on proprietary products. Standards make it easy for new competitors to enter the industry and make it easy for customers to switch from one competitor's product to another.</u> The competition stimulates new ideas for products and new ways to manufacture them more efficiently."

"Japanese software products are starting to beat American software products in Japan for the following reasons:

1. They are comparable in functional capability to the best American products.

2. They are of much higher quality than American software

3. 3-to-1 productivity advantage over the United States in software development

4. 20:1 to 200:1 quality advantage

5. Japanese emphasize management and process; US tends to emphasize technology (looking for the "silver bullet").

6. Japanese software managers stay technically up-to-date, and strive to understand software development at a detailed technical level; US managers appear more financially oriented."

"End Result:

- Quality figures are quoted for Japanese software of 8 defects per 1 million lines of released software – this is recording all problems, not just customer – reported defects

- IBM Japan produces software which has an order of magnitude fewer defects than that produced by IBM US and IBM France

- The low end of Japanese software productivity is at the high end of US companies production"

Managing Real-Time Ada

Carol A. Mattax

Hughes Aircraft Corp., Radar Systems Group

MANAGING REAL-TIME Ada

(A COMMON-SENSE APPROACH)

RICIS '90

HUGHES

RADAR SYSTEMS GROUP

C.A. MATTAX, MANAGER
SOFTWARE DESIGN & DEVELOPMENT
PROCESSOR DIVISION

MANAGING REAL-TIME Ada

HUGHES

- Ada OFFERS THE ABILITY TO IMPROVE SOFTWARE PRODUCTS IN THE "ILITIES":
  - RELIABILITY
  - MAINTAINABILITY
  - PORTABILITY
  - SUPPORTABILITY
  - QUALITY
- THIS PRESENTATION WILL FOCUS ON THE MANAGEMENT PROCESS RATHER THAN THE TECHNICAL MERIT OF THE PRODUCTS
  - PRODUCT IMPROVEMENT BY THE USE OF Ada IS ASSUMED INHERENT IN CHOOSING AND USING THE LANGUAGE

MANAGING *REAL-TIME* Ada

HUGHES

- THE REAL-TIME SOFTWARE UNDER DISCUSSION IS EMBEDDED OPERATING SYSTEMS FOR HUGHES MODULAR PROCESSORS, AVIONICS COMPUTERS SUPPORTING MULTI-SENSOR DATA AND SIGNAL PROCESSING
  - DATA PROCESSING TARGETED TO INTEL i80960 32-BIT JIAWG STANDARD
- HARD REAL-TIME CONSTRAINTS
- PERFORMANCE REQUIREMENTS DEFINED AT HIGH LEVEL THEN ALLOCATED DOWN AS TIMING "BUDGETS"
  - OPERATING SYSTEM "BUDGET" DEPENDS ON APPLICATION USAGE; DIFFICULT TO ACCURATELY QUANTIFY
  - EVEN WITH WELL-DEFINED TIMING CONSTRAINTS, IT'S NEVER FAST ENOUGH! EVERY MICROSECOND SAVED REPRESENTS POTENTIAL ADDED FUNCTIONALITY

MANAGING REAL-TIME *Ada*

HUGHES

- THE TRADITIONAL RESPONSE TO HARD REAL-TIME CONSTRAINTS, ESPECIALLY IN AN EMBEDDED OPERATING SYSTEM, IS ASSEMBLY LANGUAGE
- THE HUGHES MODULAR PROCESSOR OPERATING SYSTEM IS WRITTEN IN Ada
  - FIRST GENERATION IN Ada DUE TO DoD MANDATE
  - SUBSEQUENT GENERATIONS IN Ada DUE TO BENEFITS IN PROCESS AND PRODUCT
- TRANSITIONING FROM ASSEMBLY LANGUAGE TO Ada IS NOT EASY
  - FIRST GENERATION USED "BRUTE FORCE" APPROACH
  - IN SUBSEQUENT GENERATIONS, MANAGEMENT PROCESS TAILORED TO LEVERAGE OFF Ada

CONSEQUENCES OF "BRUTE FORCE" APPROACH TO Ada

HUGHES

- COMPILER PERFORMANCE WAS MUCH WORSE THAN EXPECTED, ESPECIALLY USING CERTAIN CONSTRUCTS
  - REAL-TIME PERFORMANCE WAS SIGNIFICANTLY DEGRADED
- RUN-TIME SYSTEM FUNCTIONALITY AND PERFORMANCE WERE INSUFFICIENT FOR REAL-TIME DEMANDS
- LEARNING CURVE FOR Ada HAS TO BE FACTORED IN
- BAD FORTRAN CAN BE WRITTEN IN ANY LANGUAGE
- SUBSTANTIAL OPTIMIZATION WAS REQUIRED TO ACHIEVE PERFORMANCE GOALS
  - INITIAL RELEASE WAS 3 TO 10 TIMES TOO SLOW

BRUTE FORCE APPROACH WORKS BUT IS PAINFUL AND INEFFICIENT

TAILORING THE MANAGEMENT PROCESS FOR Ada: REQUIREMENTS

HUGHES

- ALLOCATING PERFORMANCE REQUIREMENTS TO DETAILED TIMING BUDGETS IS A CRITICAL ACTIVITY IN SPECIFYING REQUIREMENTS FOR REAL-TIME SYSTEMS
  - TO ALLOCATE TIMING REQUIREMENTS, THE PERFORMANCE OF COMPILED CODE MUST BE KNOWN, BUT TYPICALLY ONLY AVERAGE PERFORMANCE OVER A NARROW SET OF BENCHMARKS IS KNOWN, IF THAT

- COMPILER EVALUATION AND BENCHMARKING IS REQUIRED PRIOR TO OR DURING THE REQUIREMENTS PHASE
  - EVALUATION CRITERIA INCLUDE EFFICIENCY, CODE EXPANSION, ROBUSTNESS, IDIOSYNCRACIES IN IMPLEMENTATION OF Ada, ETC.
  - VARIETY OF BENCHMARKS ARE USED:
    - STANDARD PIWG, ETC.
    - BENCHMARKS REPRESENTATIVE OF THE REAL-TIME APPLICATION AND/OR THE MOST SEVERE CONSTRAINTS

TAILORING THE MANAGEMENT PROCESS FOR Ada: DESIGN

HUGHES

- ONE OF THE BENEFITS OF Ada IS MOVING DEVELOPMENT ACTIVITIES FROM INTEGRATION TIME TO DESIGN TIME
  - USE PACKAGE SPECS TO DEFINE CSC'S AND TO UNAMBIGUOUSLY DEFINE INTERFACES
  - TEST AT DESIGN TIME BY COMPILATION RATHER THAN AT INTEGRATION TIME BY TESTING AND REWORK
    - CONFIGURE PACKAGE SPECS EARLY
    - FLOW DOWN TIMING BUDGETS AND IDENTIFY CRITICAL COMPONENTS
    - RAPID PROTOTYPING SELECTED CRITICAL AREAS PROVIDES EARLY MEASURE OF WHETHER TIMING BUDGETS ARE ACHIEVABLE AS WELL AS VALIDATION OF BENCHMARK RESULTS
    - REWORK AND REALLOCATION OF TIMING IS THUS POSSIBLE MUCH EARLIER IN THE DEVELOPMENT CYCLE

TAILORING THE MANAGEMENT PROCESS FOR Ada: DESIGN (CONT'D.)

HUGHES

- SOFTWARE ENGINEERING PRACTICES SAY IF YOU SPEND MORE TIME DESIGNING, INTEGRATION GOES FASTER, WITH LESS REWORK, AND THE PRODUCT IS BETTER.

- ESPECIALLY IN REAL-TIME SYSTEMS, WHERE THERE IS A LEGITIMATE FEAR THAT THE SYSTEM WILL FAIL TO MEET REAL-TIME CONSTRAINTS, THERE'S A PUSH TO GET TO THE LAB AS SOON AS POSSIBLE TO SEE HOW BAD PERFORMANCE IS.

- TAILORING THE PROCESS TO SUPPORT Ada FORCES MORE TIME TO BE SPENT IN DESIGN
  - CORRESPONDING SUCCESS IN INTEGRATION HAS BEEN ACHIEVED
  - THE FEAR IS STILL THERE. GETTING AN EARLY HANDLE ON TIMING AS DESCRIBED ABOVE HELPS MITIGATE SOMEWHAT, BUT THE FEAR NEEDS TO BE MANAGED AS WELL

TAILORING THE MANAGEMENT PROCESS FOR Ada: CODING

HUGHES

- THE DISTINCTION BETWEEN DESIGN AND CODE IS BLURRED WITH Ada, ESPECIALLY IF Ada CONSTRUCTS AND Ada AS PDL ARE USED TO DESCRIBE THE DESIGN. NONETHELESS, THERE'S A CODING JOB TO DO.

- FOR A TYPICAL REAL-TIME SYSTEM, WHERE EVERY INCREASE IN PROCESSOR OR COMPILER PERFORMANCE REPRESENTS MORE FUNCTIONALITY, THE NON-DETERMINISTIC FEATURES OF Ada ARE A PROBLEM.
  - WE STATICALLY ALLOCATE MEMORY, DO NOT USE RUN-TIME ELABORATION OR RENDEZVOUS, ETC. IN THE OPERATING SYSTEM

- IN ADDITION, FOR A GIVEN TARGET AND COMPILER, CERTAIN Ada CONTRUCTS MAY BE TOO SLOW FOR EFFICIENT REAL-TIME PERFORMANCE. SUCH CONSTRUCTS ARE IDENTIFIED DURING THE BENCHMARKING PROCESS

- ALL SUCH RESTRICTIONS ARE DOCUMENTED IN THE CODING STANDARD OR GUIDELINE

TAILORING THE MANAGEMENT PROCESS: INTEGRATION

HUGHES

- PLAN IN TIME DURING THE INTEGRATION PHASE FOR OPTIMIZATION
  - IT WON'T BE FAST ENOUGH!
- DEVELOP TOOLS TO TIME AND BENCHMARK SYSTEM PERFORMANCE PRIOR TO INTEGRATION
  - FOLKLORE AS TO WHERE THE TIME GOES IS OFTEN WRONG
  - SOMETIMES POOR PERFORMANCE IS DUE TO A CODING ERROR
- BENCHMARK AND DOCUMENT PERFORMANCE WITH EVERY SIGNIFICANT REBUILD TO AVOID TIMING BUILD-UP AGAIN
- AVOID THE TEMPTATION TO USE ASSEMBLY LANGUAGE EXCEPT WHEN IT'S REALLY THE LAST RESORT
  - CAN COVER UP ERRORS, POOR DESIGN, OR POOR IMPLEMENTATION WHICH COULD HAVE BEEN CORRECTED USING Ada

TAILORING THE MANAGEMENT PROCESS FOR ADA: DOCUMENTATION

HUGHES

- DOCUMENTATION IS A SIGNIFICANT SOFTWARE DEVELOPMENT ACTIVITY FOR DoD SYSTEMS
- THE DOCUMENTATION PROCESS AND PRODUCT CAN BE SIGNIFICANTLY IMPROVED BY LEVERAGING OFF Ada:
  - IRS & IDD: USE Ada PACKAGE SPECS AUGMENTED BY COMMENTS
  - USER'S MANUAL, AT LEAST FOR OPERATING SYSTEMS: START WITH USER SPEC WITH COMMENTS AND AMPLIFY AS DEVELOPMENT CONTINUES
  - DESIGN DOCUMENTATION: USE PACKAGE SPECS AND Ada AS PDL; SUPPLEMENT WITH DATA FLOWS, ETC.
  - AS-BUILT DOCUMENTATION: REVERSE ENGINEER FROM THE CODE TO ENSURE ACCURACY; SUPPLEMENT AS NEEDED

MANAGING REAL-TIME Ada

HUGHES

- Ada AND REAL-TIME ARE NOT INCOMPATIBLE, BUT GREAT CARE MUST BE TAKEN TO:
  - UNDERSTAND THE COMPILER PERFORMANCE
  - MANAGE THE DEVELOPMENT PROCESS TO LEVERAGE OFF Ada
  - MANAGE THE FEAR OF NONPERFORMANCE TO HARD REAL-TIME REQUIREMENTS

Session 2

Software Engineering Activities at SEI

Chair: **Clyde Chittister,** *Program Director of Software Systems, Software Engineering Institute, Carnegie Mellon University*

Carnegie Mellon University
Software Engineering Institute

Software Systems Program

November 8, 1990

RICIS "90"

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Sponsored by the U.S. Department of Defense

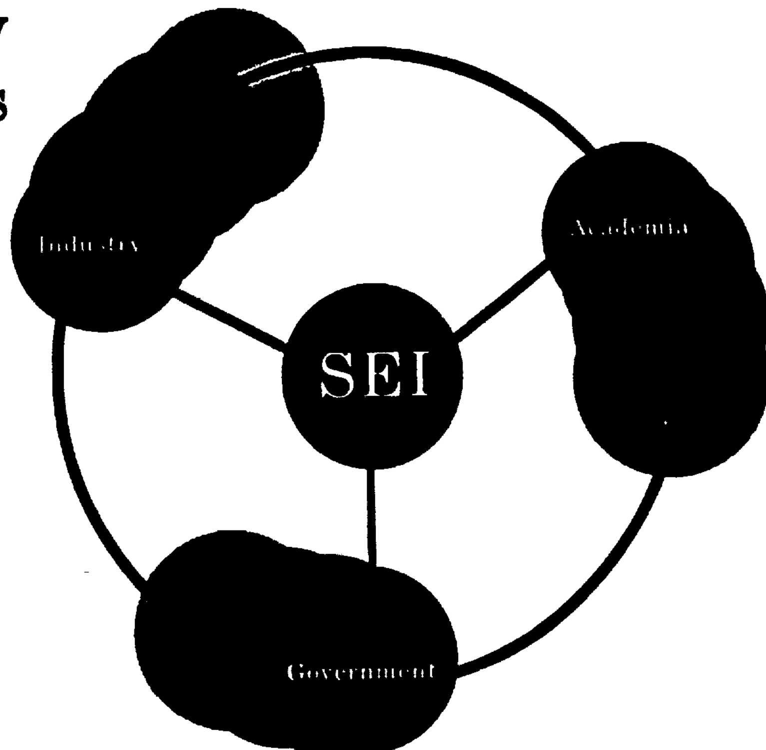
SSP080390

SEI Mission

Provide leadership in advancing the state-of-the-practice of software engineering to improve the quality of systems that depend on software.

Carnegie Mellon University
Software Engineering Institute

Technology Flow Paths



Purpose:

To facilitate a higher quality communication

Software Systems Program Objective

Assist the MCCR community in improving the way software is developed for real-time distributed systems

- integrate software and systems engineering
- increase the effective use of technology
  - Ada
  - design methods
  - common architectures
  - scheduling algorithms
- Reduce the risk of adopting new technology

Strategy

Identify and select key technical issues to investigate.

Select application domains in which to work.

Establish relationships with influential customers and vendors in these domains.

Evaluate and prototype potential solutions to selected technical problems.

Conduct proof-of-concept experiments in selected application domains.

Facilitate the introduction of these concepts into practice.

Software Systems Projects

Rate Monotonic Analysis for Real-Time Systems

Software for Heterogeneous Machines

User Interface - SERPENT

Real-Time Embedded Systems Testbed

Systems Fault Tolerance (proposed)

Real-Time Data Management (potential)

User Interface Development
Serpent UIMS

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Sponsored by the U.S. Department of Defense

Introduction

- Problems
- Objectives
- Approach
- Serpent Architecture
- Serpent Editor
- Outside Efforts
- Status

User Interface (UI) Problems

- User interface accounts for large portion of life cycle costs
- Impacts all aspects of the life cycle
  - requirements
  - development
  - sustaining engineering

Life Cycle Problems

- Requirements
  - evolutionary, not well specified
  - written specifications inadequate
  - customers may not know what is practical
- Design/implementation
  - very labor intensive
  - inadequate existing methods and tools
- After system completed
  - frequent and complex changes required
  - difficult to take advantage of new I/O media

Objectives

- Make user interfaces easier to specify
- Support incremental development of user interfaces (prototypes)
- Provide for a "bridge" between prototype and production versions of system
- Support insertion of new I/O media during sustaining engineering

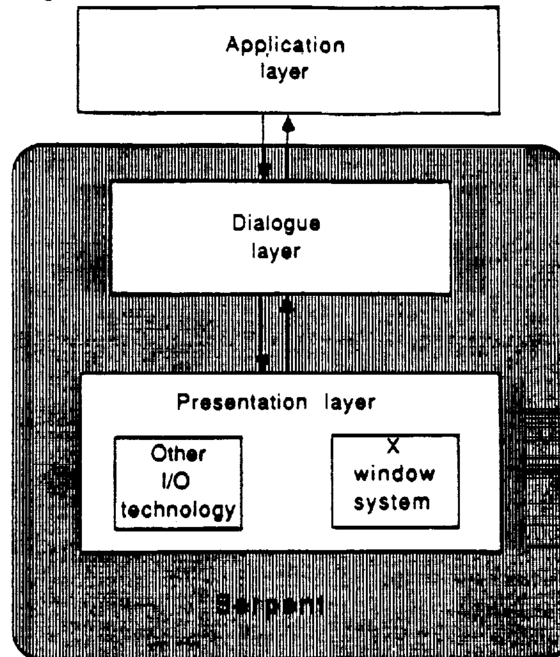
Approach to Reducing UI Problems

- Provide single tool which supports incremental specification and execution of interface
- Separate concern of user interface specification and execution from rest of system concerns
- Apply non-procedural language and graphical techniques to user interface specification

Serpent UIMS

- Has specialized language for user interface specification
- Supports I/O media independent applications
- Supports both prototyping and production
- Supports multiple I/O media for user interactions
- Supports ease of insertion of new I/O media

Serpent Architecture



Slang, UI Specification Language

- Based on production model
  - data driven
  - allows multiple threads of control
- Provides multiple views of the same data
  - implemented with constraint mechanism
  - re-evaluates dependent values automatically when independent values modified
  - applies to application values, I/O media display values, and local variables

Prototyping

- Detailed knowledge of Serpent dialogue model is not required
- Application not required
- Slang allows definition of local data
- Serpent automatically enforces constraints
- Reasonably sophisticated prototypes can be generated, e.g., visual programming

Input/Output Media

- Serpent designed to simplify the integration of I/O media
- Currently Integrated
  - digital mapping system
  - X11 Athena widget set
- Integrations anticipated/in progress
  - Motif
  - Open Look

Application

- Can be written in C or Ada
- Views Serpent as similar to database management system
- Creates, deletes, or modifies data records
- Informed of creation, deletion, or modification of data records by dialogue layer

Serpent Editor

- Layouts of user interface are best specified or examined graphically
- Logic, dependencies, and calculations are best specified textually
- Serpent Editor has two portions
  - graphical part for examination and specification of layout
  - structure part for textual specification
- Implemented using Serpent

Outside Efforts -- ARMY TO&P

- FATDS/CECOM - on contract
  - Port Serpent to ATCCS CHS
  - Install Serpent at Center for Software Engineering
  - Technical support to Magnavox
- FAAD - preliminary negotiations underway
  - Technical support to TRW

Outside Efforts -- Standardization Work

- IEEE P1201.3
- OSF
- Unix International
- UIMS Working Group

Outside Efforts -- Commercialization

- Dedicated Company
- Consortium
- Multiple H/W and/or S/W vendors

Status

- Serpent (with visual portion of editor) in alpha test
- Supported for Sun, VAX (Ultrix), DECStation, HP (HPUX)
- Beta version of Serpent (including complete editor) available 4QCY90

Session 3

Software Reuse

Chair: **Robert Angier**, *IBM Corp.*

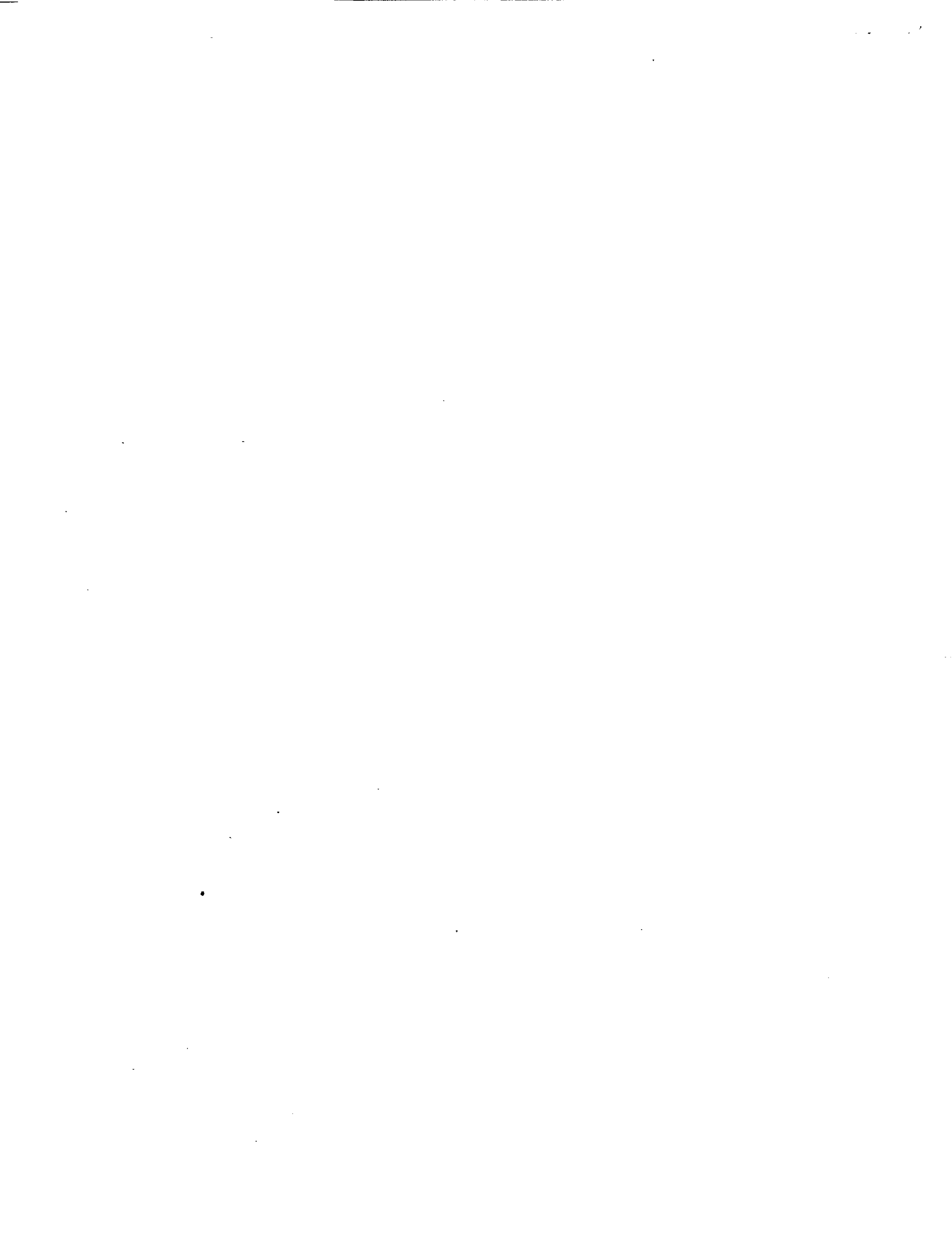
Research Directions in Software Reuse

November 8, 1990

Will Tracz

MD 0210
IBM Federal Sector Division
Owego, NY 13827
(607) 751-2169
net: OWEGO@IBM.COM

Unclassified



Where Does Reuse Start?

Will Tracz

MD 0210
IBM System Integration Division
Owego, N.Y. 13827
OWEGO@IBM.COM or TRACZ@[illegible]RA.STANFORD.EDU
/+ALLA

Preface

The following is a transcript of the keynote address for the Reuse in Practice Workshop sponsored by IDA, SEI and SIGADA. The workshop was held in Pittsburgh, PA at the Software Engineering Institute, July 11-13th, 1989. The goal of this talk was to establish some common vocabulary and to paint a broad picture of the issues related to software reuse.

Overview

Software reuse is the type of thing some people swear by. It is also the type of thing that some people swear at. Software reuse is a religion, a religion that all of us here today pretty much have accepted and embraced. The goal of this talk is to question the foundation of our faith - to test the depth of our convictions with the hope of shedding new light on our intuitions. I do not claim to have experienced divine intervention. You don't need to take what I say as gospel truth. I believe in what I say, but what you hear may be something different. Again, let me encourage you to disagree - to challenge the position I have taken on the issues I will be presenting. Before I proceed further, I need to qualify software reuse by providing a definition.

Software reuse, to me, is the process of reusing software that was designed to be reused. Software reuse is distinct from software salvaging, that is reusing software that was not designed to be reused. Furthermore, software reuse is distinct from carrying-over code, that is reusing code from one version of an application to another. To summarize, reusable software is software that was designed to be reused. The major portion of my talk will focus on examining the rhetorical question, "Where does reuse start?"

Introduction

If I were to ask you, "Where does reuse start?", your reply might be, "What do you mean? That seems like a pretty vague and nebulous question!"

I agree, so I have done a little top-down stepwise refinement and broken the question up to focus on three areas - the three P's of software reuse: *product*, or what do we reuse, *process*, or when do we apply reuse, and finally *personnel*, or who makes reuse happen. I guess I could have called it the three W's of reuse: *what*, *when*, and *who*.

"Why is this an important question?" you might ask. The first answer that comes to *my* mind is that if you would like to build a tool to help reuse software, it would be reasonable to know: 1) what you were trying to reuse, 2) when you would be doing it, and 3) who would be using it. That is one reason, a pretty good reason, but not the only reason for asking the question "Where does reuse start?" Rhetorically, if one could understand the ramifications, implications and economic justifications of the answer to the original question, "Where does reuse start?", one would better be able to answer the question "Where should reuse start?" and "What needs to be done to make it happen?" This is the real question I think we are here to answer.

Product

If one examines the question of "Where does reuse start?" by focussing on the products being reused, one could ask "Does reuse *start* with code?" There is no denying that software reuse generally *ends* with "code". But, this still is a pretty broad statement. After all, code could be source code, object code, a high level language statement, a function, a procedure, a package, a module, or an entire program. The issue raised then is "What is the granularity of the code that you want to reuse?" The larger the granularity, the larger the "win" is in productivity. The overhead for finding, understanding and integrating a reusable software component needs to be less than designing and

ORIGINAL PAGE IS
OF POOR QUALITY

writing the code from scratch. This supports the argument for the reuse of higher granularity objects such as software packages, modules or classes.

Just as we could debate the granularity of the object being reused, one could argue about the level of abstraction that is being manipulated. Does reuse start with a **design**? A design is a higher level abstraction compared to an implementation. Let me emphasize that the advantage of starting reuse from a design is that a design is at a higher level of abstraction than an implementation. Or, in other words, a design has less implementation details that constrain its applicability.

This brings out a point made in a recent paper I have been writing called "Software Reuse Rules of Thumb". In it I propose two general rules of thumb for software reuse: 1) to separate context from content and concept, and 2) to factor out commonality, or to rephrase this second rule a bit, to isolate change. If one applies the first rule of thumb, a program design, say at the detailed logic level, should have absent some (but not all) of the contextual information that will be supplied at implementation time. That is, the implementation issues, such as specific operating system or hardware dependencies, are neither part of the content, which is the algorithm or data flow nor part of the concept, which is the functional specification. I will address the second rule of thumb, factoring out commonality, later.

Before proceeding, I would like to emphasize the importance of representation, especially from a tool perspective. Remember I stated earlier that one of the reasons for looking for an answer to the question of "Where does reuse start?" was to provide a rational for building tools to assist in the reuse process. This implies that we would like a machine manipulable reusable design representation. This is not easy! But, I believe the state of the art is now evolving to a point where there are results of software reuse starting from design. The projects, that I am aware of, have been at MCC, with the DESIRE system, and at Toshiba, where in the 50 Steps per Module system, they are working on an expert system to automatically generate C, FORTRAN or Ada from low-level design data-flow charts. Furthermore, they claim success in reverse engineering existing software by synthesizing data-flow diagrams for potential reuse.

Continuing our analysis of the question "Where does reuse start?", could reuse start with a program's **specification**? By specification, I mean a statement of "what" a program need's to do, not "how" it is supposed to do it. There is a simple answer, yes, in limited contexts, program specifications can be reusable. But research in automatic programming tells us that this is a hard problem to extrapolate outside of narrow domains.

Speaking from personal experience, we at IBM in Owego have developed some reusable avionics specifications. When I say specifications, I mean MIL-STD-2167 System Requirements Specifications (SRS). They are highly parameterized documents full of empty tables and missing parameter values. The systems analyst, in effect, programs a new module by specifying the values in the tables of the SRS document. An application generator then reads the document and builds the data structures necessary to drive the supporting software.

Completing the waterfall model, we can ask the question on whether reuse can start with a problem definition (**requirements**). This is an interesting question. One might ask how? One could reason that if the same requirements can be identified as being satisfied by certain previously developed modules, then clearly those modules are candidates for reuse. Well that is a big if. It is significantly dependent on the traceability of requirements to specifications, the traceability of specifications to design, and the traceability of design into code and, also into **test cases**, and **documentation**.

Here is where a hypertext system's information web is ideal for linking these artifacts together. With a hypertext system, you can walk the beaten path to find out what code to reuse. But, there is a catch. As Ted Biggerstaff has repeatedly stated, there is no free lunch. You have to pre-engineer the artifacts to fit into the network, and spend the time and effort to create the links. Finally you need to somehow separate the context of the objects from the content. One mechanism for achieving this goal is through parameterization. Parameterization is a way to extend the domain of applicability of reusable software. Parameterization allows a single module to be generalized over a set of solutions.

To summarize, the issue we have been exploring related to the question of "Where does reuse start?" is really the question "What software artifact does reuse start with?" Part of the answer lies in the fact that we know that software reuse generally *ends* with the reuse of code. Where it starts depends on: 1) how much effort we want to place in developing the reusable artifact that we want to begin with, 2) how effectively we can link it to an implementation, and 3) (maybe not so obvious) how effectively we generalize the implementation.

There is a fourth dependency having to do with the process of software reuse. This is topic I will address subsequently. First I would like to reflect on the generalization issue of an implementation. One must rec-

ognize that as we progress down the waterfall model, from requirements to implementation, each artifact adds more detail. An implementation is one instantiation of a design. There could be several implementations of a design just as there could be several designs that satisfy a specification but that have different performance and resource attributes. The key is factoring out the commonality by separating the context from the concept and content. The concept becomes the functional specification. The content becomes a template or generic object. The context becomes possible instantiation parameters. We have identified some of the dimensions and implications related to which software artifact to start reuse with. I have concluded that code is a safe place to start and is, in most cases, the place one ends up. I also have mentioned that hypertext is the way to establish the traceability between requirements, specification, design, tests and implementation.

Process

Turning to the software development process, one could observe that most software reuse starts at the **implementation** phase. One could modify the software development process to include a step where, at implementation time, one would look for existing software to save having to write new code that would do the same thing. With a little luck, this usually works. But with a little foresight, this usually works better. How often is it the case that the code one wants to reuse has to be modified because either it was not implemented to exactly fit the new context it is being reused in, or it was not implemented to provide a parameter for adapting it to a different context, or the design was such that it placed unnecessary constraints on the implementation? If the software designer had not placed the (somewhat) arbitrary design constraints, then the implementation could be used as is.

Therefore, with a little foresight, reuse might better start at **design** time. The implementer could then leverage off the functionality of existing implementations. This is where the bottom-up aspect of reuse meets the top-down functional decomposition aspect of most design processes. One could argue that object-oriented design would eliminate this problem. Let me say that object-oriented design helps reduce the problem of the design not meeting the implementation, but parameterization still is the key for controlling this process.

One could just as easily extend the same argument for looking for reuse opportunities at design time, for the same reasons, to the **specification** and **requirements analysis** phases of the software life cycle. Again, by identifying earlier on in the software development life cycle, what is available to be reused, trade-offs can made in the specifications, or designs can be tailored to leverage off the existing software base.

Let me now introduce somewhat of a new phase in the traditional waterfall model that has been added explicitly to support software reuse. I define **domain analysis** to be a generalization of requirements analysis - instead of analyzing the requirements for a specific application, the requirements of a generic application are quantified over a domain. Applying my two rules of thumb: commonality is factored out and context is separated from concept and content. Reusable objects are identified, and their context defined.

If one recognizes that the software development life cycle needs to be modified in order to inject software reuse technology, then, relating to personal experience, reuse opportunities and potential can be identified at **code review time**, or at **design review time**. If one looks at the Programming Process Architecture used in IBM, one can see these criteria called out as being integral parts of the inspection process.

But then again, instead of reuse being addressed during the software development effort, maybe reuse could start as an after thought (**project follow-on**). After one pass through the software development life cycle, the second time through one can begin to see the commonality between applications. Quoting Ted Biggerstaff's rules of three "If you have not built three real systems in a particular domain, you are unlikely to be able to derive the necessary details of the domain required for successful reuse in that domain."

As a side point, there is a second rule of three. "Before you can reap the benefits of reuse, you need to reuse it three times." The empirical evidence I have seen to date bear this out.

A better choice for where reuse should start is at the beginning of a project (**project start up**). Here, the software development process can be defined, reusable software libraries can be set up and standards as well as tools developed.

To share with you again my personal experience, in one large Ada project, A Computer Integrated Manufacturing (CIM) effort involving 350K SLOCS, the project had a PRL - Project Reuse Lead. He was responsible for sitting in on all design and specification reviews to identify commonality between subsystems and support the communication and application of reuse technology. Because of software reuse, factoring out commonality, the size and development effort of the project was reduced by over 20%. This

ORIGINAL PAGE IS
OF POOR QUALITY

is a successful example of where reuse started at the beginning of a project.

But, then again, maybe reuse could start at the end of a project (**project wrap-up**). I am reminded of the General Dynamics approach for developing reusable software related to an early version of the DARTS system. Here, after a project was completed, and before the design and development team was assigned to a new project, they locked everyone up in a room and wouldn't let them out until they developed an archetype of the system. That is, they recorded how and what to modify in the system so that it could be reused in the future.

While this is one approach for developing reusable software, it seems like putting the cart in front of the horse. But, then again, it is reasonable, upon the completion of any project to identify likely components to add to a reuse library.

Finally, we are all in this for the bottom line. Let me state my version of the Japanese software factory's motto: "Ask not what you can do for your software, but what your software can do for you." It makes sense, dollars and cents, to capitalize on existing software resources and expertise. But, you need to develop a **business case** to justify the additional cost of developing reusable software.

To summarize, the issue we have just explored related to the question of "Where does reuse start?" is really the question "Where in the software development life cycle does reuse start?" Where it starts depends on 1) how one modifies the software development process to identify opportunities for reuse, and 2) how one either modifies or extends the software life cycle to identify objects to make reusable. The bottom-line is that software reuse is a good example of software engineering discipline.

Personnel

Turning to the last dimension I identified related to the question of "Where does Reuse Start?", we will focus on the key players in the reuse ball game. The first player to come to bat is the programmer. Does reuse start with a **programmer**? Most programmers are responsible for the design and implementation of software. If they can identify a shortcut to make their job easier, or to make them appear more productive to their management, then they probably will be motivated to reuse software. But, while programmers might be inclined to reuse software if it was fun, or it was the path of least resistance, or if they are told to, the real issue is "Who is going to create the software to reuse in the first place?" There needs to be a critical mass of quality software for programmers to draw upon in order for them to fully subscribe to the reuse paradigm! So, how do we bootstrap the system?

Maybe **managers** can instill a more altruistic attitude on their programmers. This, of course, becomes a question of budget cost and schedule risks associated with the the extra time and effort needed to make things reusable.

Reuse is a long term investment. Maybe the expense of developing reusable software should be spread across a **project**! With reuse raise to the project level, there would higher potential for a larger return on investment, plus more insight and experience in prioritizing what should be made reusable. Again, there is no free lunch, A project manager would have to authorize the cost. But project management is generally rewarded for getting a job done on time and under budget. There is no motivation for making the next project look good. This shortsightedness needs to be resolved with **top management**.

Indeed, this is the case, both here and abroad. At NTT, GTE, IBM, TRW, to name a few companies, reuse incorporation and deposition objectives are being set. For instance at NTT, top management has set a reuse ratio goal of 20% on all new projects, with a deposition ratio quota of 5%. That is, all new programs ideally should consist of at least 20% source code from the reuse library and all new programs should try and deposit at least 5% of their source code to the reuse library (subject to the acceptance guidelines, constraints, and ultimate approval of the Reuse Committee).

But, upper management edicting reuse to happen doesn't insure success. That is why there is a strong argument for reuse to start in the classroom (**educator**). The education system, while it is good at teaching theory, might embrace a little more of the engineering discipline and teach software building block construction or composition of programs. Courses are needed in domain analysis, application generator construction, and parameterized programming, as well as the availability of pre-fabricated, off-the shelf components structured to facilitate the construction of new applications in a classroom setting. Again, critical mass is needed to bootstrap the system.

Besides the reuse mind set, maybe reuse should start with a tool set (**tool developer**). Personally, I do not see the need for exotic and elaborate tools to support reuse. Although, I am biased towards using a multimedia hypertext system for the capture and representation of domain knowledge, which I consider crucial to understanding what and how to reuse software.

ORIGINAL PAGE IS
OF POOR QUALITY

Have I run out of people who possibly could start the reuse ball rolling? Have I saved my heavy hitters for last? Should reuse start with the **customer**? It depends on the customer! A large customer, like the Department of Defense, could easily demand certain reuse requirements be met. Of course, there might be a small initial overhead cost associated with getting the ball rolling, but once the system was primed, once application domains were populated with certified, parameterized, well documented, reusable components, then long term benefits could be reaped.

I have added the **salesperson** to this list of individuals who could play a role in determining where reuse might start. The reason is that if a salesperson knows the marketplace and knows potential customers, then they could play a key role in building the business case necessary to justify the capitalization of software for reuse.

Finally, I have added the **systems analyst** as being a person who possibly could be instrumental in starting software reuse. I admit, he joined the team late, but he turns out to be a clutch player. Back to the issue of putting the horse in front of the cart. Before you can reuse software, you need software to reuse. Who are you going to call? The domain analysts! Who are the most qualified individuals in an organization to 1) analyze a problem domain, 2) determine logical subsystems and functions, and 3) determine the contents or requirements of modules and anticipate the different contexts that they might be applied under? The systems analysts. They have made life so difficult for some of us programmers in the past by providing incomplete or inconsistent or, worse yet, too detailed specifications. This is a wonderful opportunity to work together toward a common goal.

To summarize, the issue we have been exploring related to the question of "Where does reuse start?" has been identifying the roles played by certain individuals in an organization related to making software reuse happen. In retrospect, several of the key players had non-technical roles in the game! A point that bears distinction and should come as no surprise.

Summary

In conclusion, the goal of my presentation was to bring to light issues surrounding software reuse. To force you to question what you might have accepted on blind faith. I have probably raised more questions than I have answered, but, that is good. Hopefully it will provide you opportunities for discussion. Finally, I have shown, as a wise old owl once stated, "It is not what you know, but who, you know?" that often is necessary for success. Software reuse is no exception to this rule. Software reuse is a people issue as well as a technology issue.

ORIGINAL PAGE IS
OF POOR QUALITY