P R O C E E D I N G S
A P P E N D I C E S
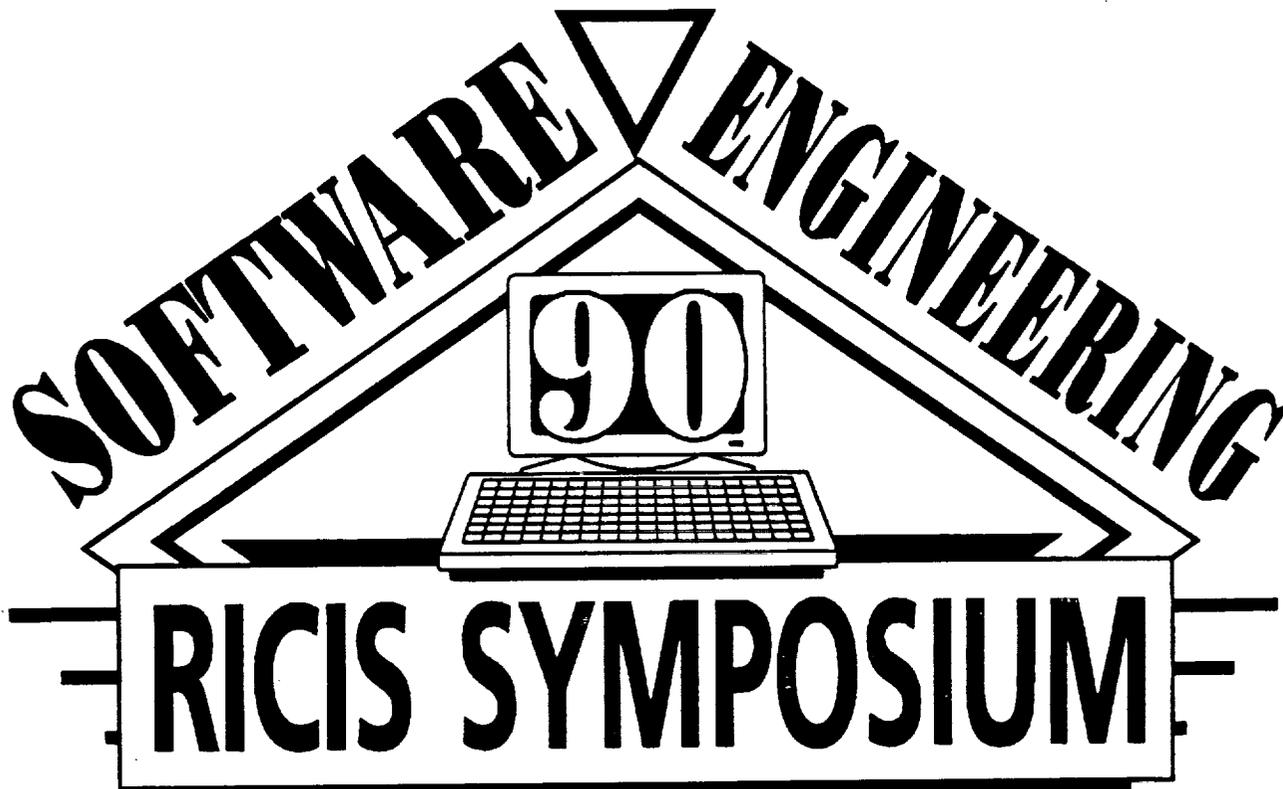
SOFTWARE ENGINEERING

90

RICIS SYMPOSIUM

Aerospace Applications & Research Directions

November 7 & 8, 1990

RESEARCH INSTITUTE FOR COMPUTING AND INFORMATION SYSTEMS

Co-Sponsored by
University of Houston-Clear Lake
NASA/Johnson Space Center

Session 1

Lessons Learned in Software Engineering

Chair: **Gary Raines**, *Manager, Avionics Systems Development Office, NASA/JSC*

Report from NASA Ada User's Group

John R. Cobarruvias
Flight Data Systems Division, NASA/JSC

NASA Johnson Space Center Flight Data Systems Division	**SUBJECT** 3rd Annual NASA Ada Users Symposium	**NAME** EK/John Cobarruvias
		DATE November 7, 1990

SYMPOSIUM STATUS

John R. Cobarruivas, Chairman
Flight Data Systems Division
NASA/JSC

AGENDA

- History
- Goals
- Evaluation
- Detailed Symposium Agenda & Status

SYMPOSIUM HISTORY

- Started in 1988
- GSFC Sponsored
- SEL Workshop
- JSC's Participation resulted in sponsoring 1990 at JSC
- Supported by MITRE and UH-CL

GOALS OF SYMPOSIUM

- Recognize achievements and current projects at the various NASA centers
- Provide a forum to exchange ideas
- Provide a forum to share experience using Ada
- Encourage communications within the NASA Ada community

EVALUATION

- Achieved our goals
- 350 registered participants
- ~450 total attendance
- All centers represented
- Canadians
- Various universities
- Received over 40 papers
- Excellent technical support

EVALUATION (cont)

- Overall comments have been positive
    - Moving the Symposium provides different perspective
    - Allows other centers to participate
    - Excellent center status reports
    - Chairman did an excellent job
    (should be a basis for a promotion)

EVALUATION (cont)

- MITRE
    - Excellent technical support
    - Understood need for a technical focus
    - Paper selection and evaluation
    - Overall session arrangement

- UH-CL
    - Excellent logisitic support
    - Overall arrangements
    - I/F with MITRE

DETAILED SYMPOSIUM AGENDA & STATUS

- 5 Sessions/5 NASA sites, 13 papers
- NASA site status
- Object-oriented methods and simulation
- CM
- Distributed systems
- Reusability
- AI

DETAILED SYMPOSIUM AGENDA & STATUS (cont)

- Key speakers
    - Ralph Crafts (Ada Strategies)
    - Jack C. Heberlig/MITRE
    - Excellent closing remarks by the chairman

- Reception

NASA SITE STATUS

- Ada alive and well (could be better)
- JSC, GSFC, LaRC, LeRC, JPL
- Significant Ada Development
- No longer prototyping and research
- Scheduled delivery of Space Certified Ada S/W

JSC

- SSE (Rational Initiative)
- Major Ada Initiatives
- DMS
- ITVE
- SIB
- MSC
- TSC

GSFC

- Tremendous increase
- 1985 5 Staff years
- 1990 200 S.Y.
- Flight Telerobotic Servicer
- TDRSS
- EUVE Co-processor Flight Software
- HST
- Continued support of SEL

LaRC

- Currently 1 Branch using Ada
- 11 Projects
- Other Branches using Ada for prototypes
- Established a Software Engineering and Ada Lab (SEAL)
- Sponsored 15 classes in 1 year
- Contractor supported

LeRC

- Significant increase
- SSFP Projects (WP4)
- Ada training a key factor
- Required additional training

JPL

- Established Ada Development Lab
- Few systems using Ada
- 1 Flight System CRAF/CASSINI
- But...interest is growing
- Most problems directly related to cost of training
- Comprehensive training program with NASA funding is required

NASA ISSUES

- 3 sites with software engineering and Ada labs
    - GSFC SEL
    - JPL ADL
    - LaRC SEAL
- Training is currently site specific, site funded
- Requires an overall NASA initiative
- Documented in "Transition to Ada Plan"
- Excellent ideas, plans
- Looking to HQ to implement the plan
- All sites supportive and moving in the right direction

LUNCHEON SPEAKER

- Ralph Crafts
- Editor, Ada Strategies
- Confirmed the NASA transition issue
- Emphasize training
- Highlighted Ada success stories
    - Stealth Bomber
- Management support of Ada

CONCLUSIONS

- Achieved our goals
- Well received by the community
- 1991 Symposium, LaRC
- Personal perception of Ada

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

The Programmers
ENVIRONMENT

- **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**

RESULTS

CASE 1

- **Triplex Digital Flight Control System**

  - **Not synchronized**

- **Analog backup**

- **Each computer samples sensors independently, uses averages of good channels**

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

    - <u>Although switch value was voted, unvoted value was used</u>

- 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

- 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

- 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

November 2, 1990 11:12 AM HTC/013

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

Present Day Tools

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

October 31, 1990 10:30 AM HTC015

- 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)

Application :

Reduced graph of the called modules of root

Language: ADA

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**

| Application : | RECURSION DETECTION | Language: ADA |
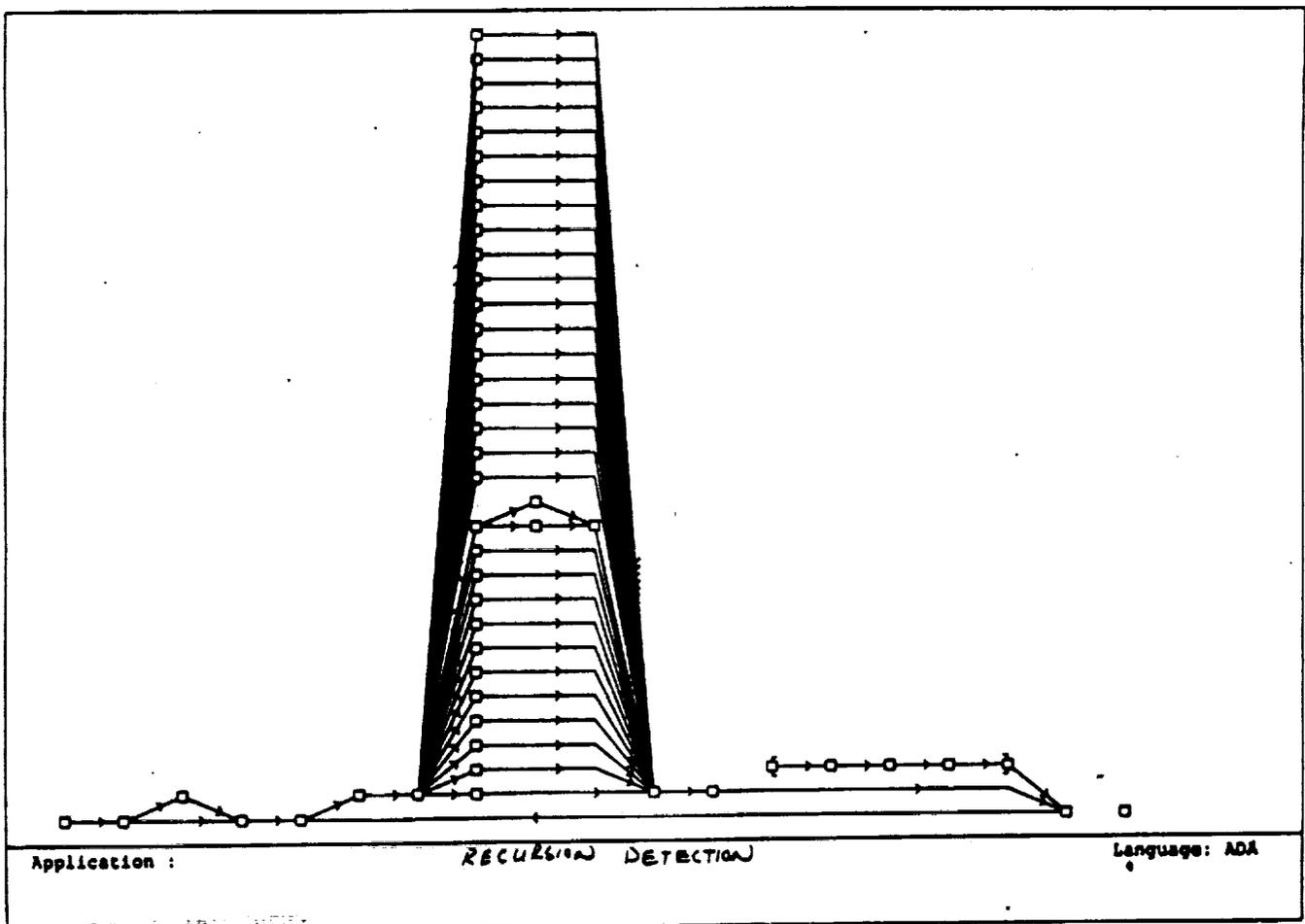
Automatic Code Generators

- **Caede**
- **Matrix**

REDUNDANT DATA BUS SOFTWARE

PROCEDURE   CONTROL FLOW   PACKAGE
DATA FLOW

71019 M3074-12

REDUNDANT DATA BUS SOFTWARE

71019 M307

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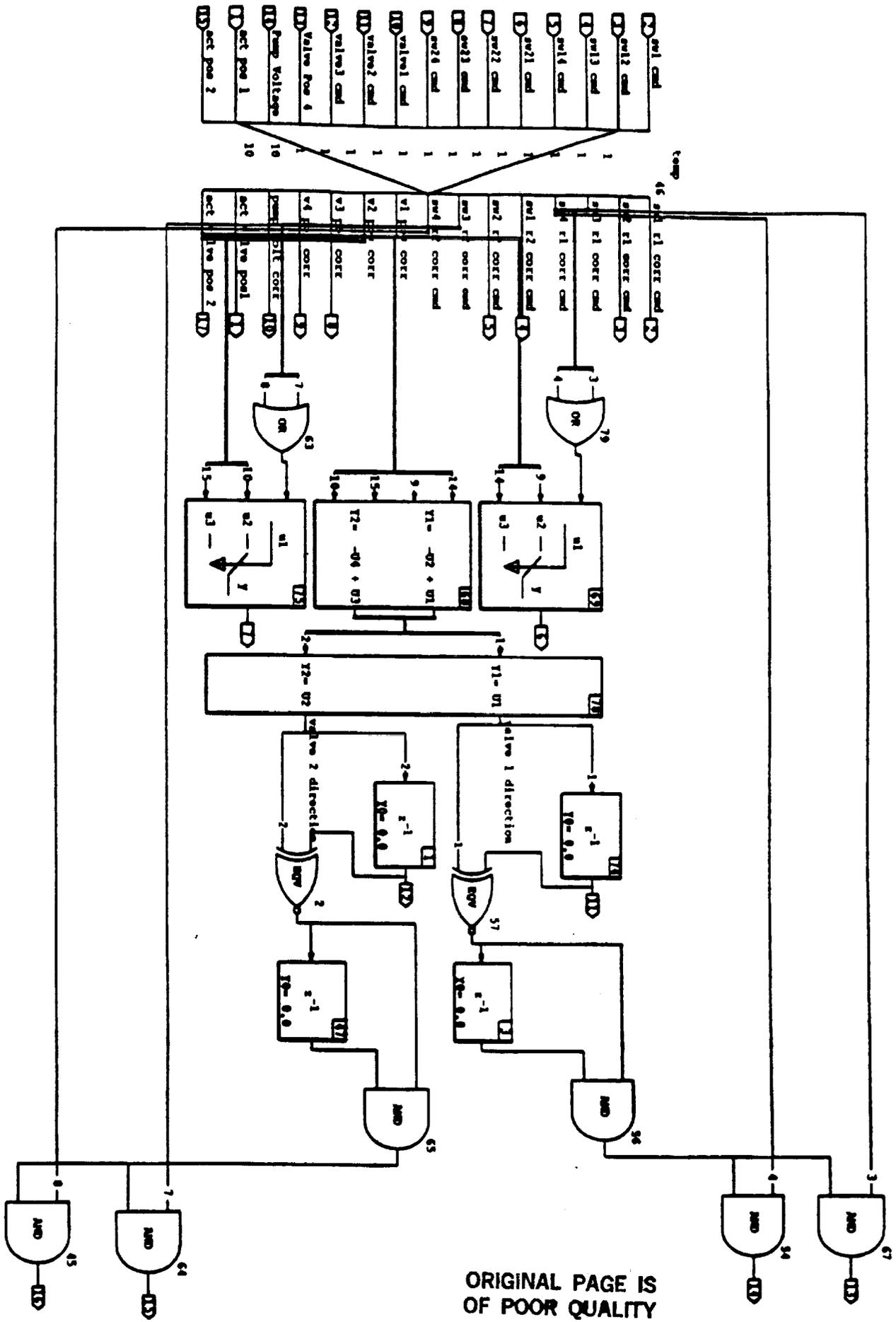
Sampling Interval   First Sample   Ext.Inputs   Ext.Outputs   Enable
1.   0.   15   17   Parent

ORIGINAL PAGE IS
OF POOR QUALITY

<u>Future</u>

- ● **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
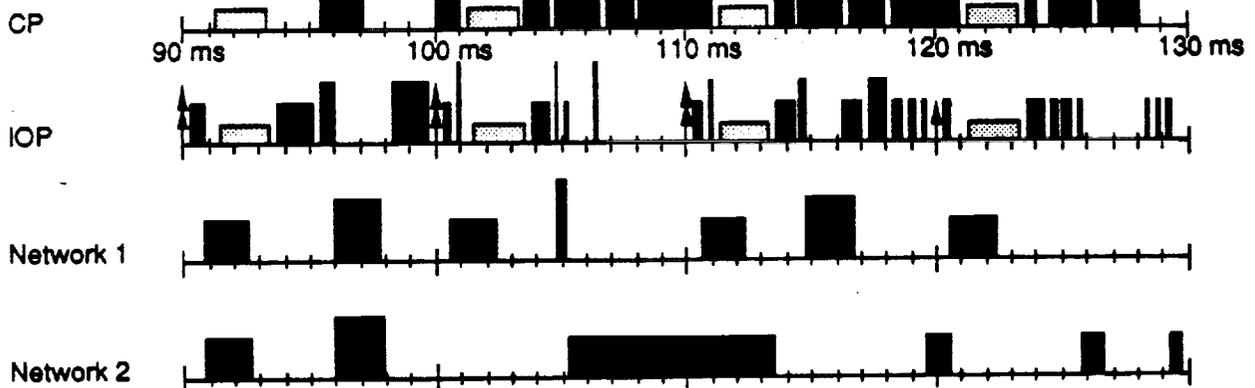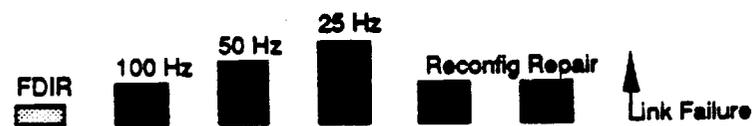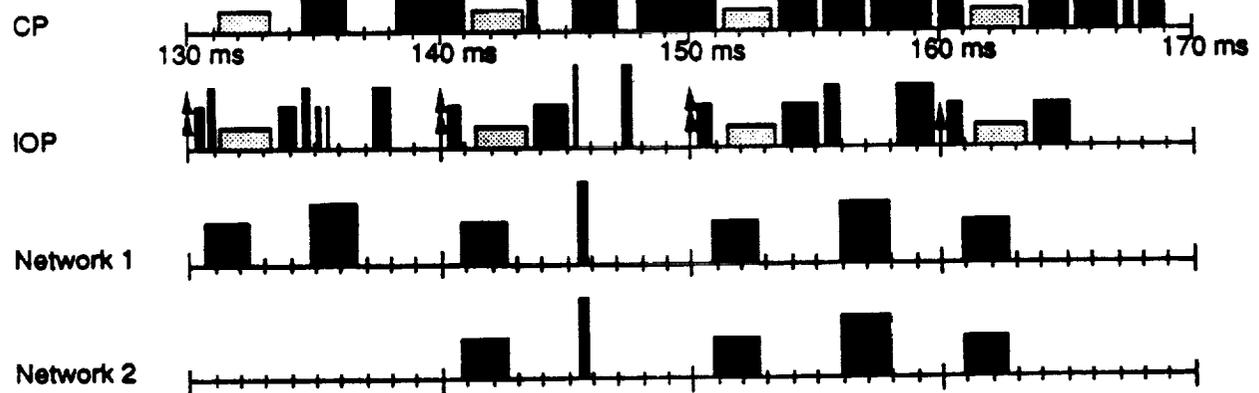
- ● **Need portability**

    - ● **Standard interfaces**

    - ● **Graphics**

    - ● **Data bases**

November 5, 1990 10:12 AM HTC025

- ● **Common software architectures**

    - ● **Exist for compilers & operating systems**

    - ● **Does not exist for application software
      (hardware years ahead in this regard)**

BOEING

- **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

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

Designer

specs   program

Syntax of program
language
- - - - - - - - - →
and specification
language

Verification
condition
generator

Semantics of
←
program language

Verification
conditions

Lemmas →

Hints →

Mechanical Theorem Prover

Semantics of
←
specification
language

Inductive →
assertions

proved   unproved   counter
example
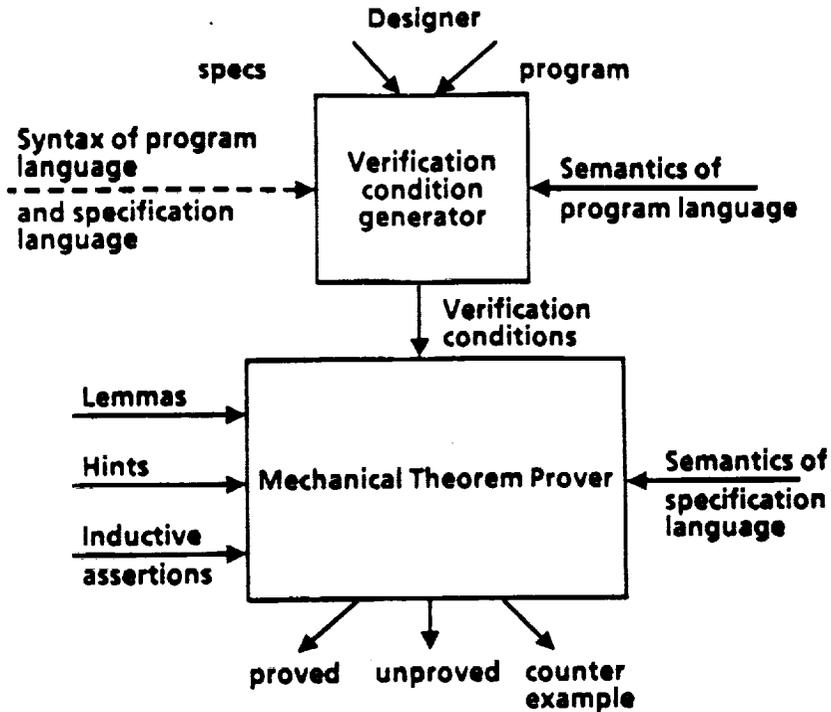
November 5, 1990 2:48 PM HTC044

- **Subset of Fortran**

- **Subset of Pascal**

- **Subset of Ada**

- **Subset of "C"**

- **Gypsy**

November 5, 1990 2:48 PM HTC043

A Growing Fear

"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."

BOEING

"Japan's electronics industry is the worlds best and largest because it is the most competitive. 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. 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."

November 5, 1990 10:42 AM HTC039

"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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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*

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

1

SSP060390

SEI Mission

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

Technology Flow Paths

Purpose:

**To facilitate a
higher quality
communication**
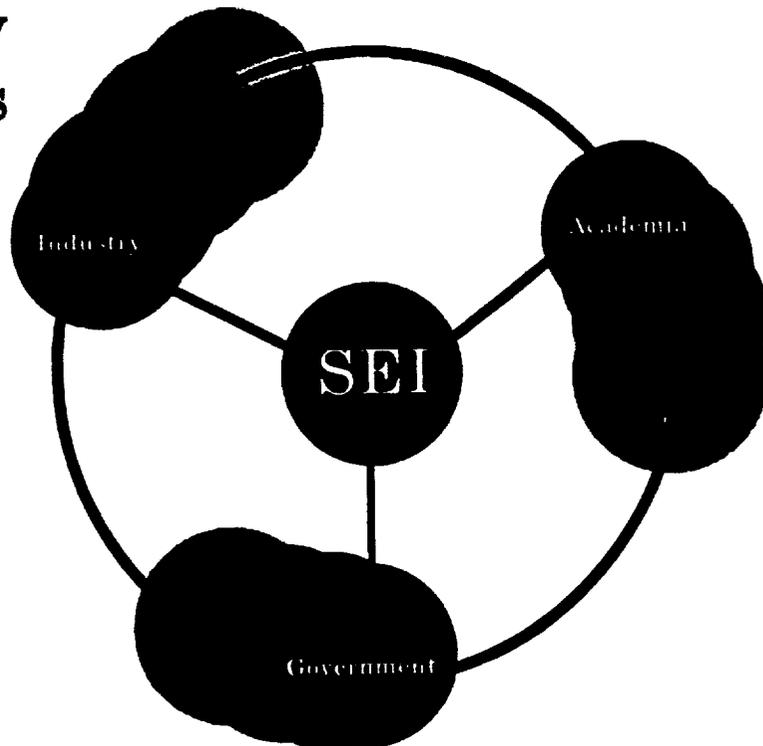
Software Systems Program Objective

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

3   SSP000300

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.

4   SSP000800

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

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

90-Serpent-reed-3

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

90-Serpent-reed-4

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

90-Serpent-reed-5

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**

90-Serpent-reed-13

Outside Efforts -- Standardization Work

- **IEEE P1201.3**

- **OSF**

- **Unix International**

- **UIMS Working Group**

90-Serpent-reed-14

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

IBM

Where Does Reuse Start?

Will Tracz

MD 0210
IBM System Integration Division
Owego, N.Y. 13827
OWEGO@IBM.COM or TRACZ@SIERRA.STANFORD.EDU
/+ ᴧ ᴧ ʜ

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

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

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 crit-

ical 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.

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.

A CONCEPTUAL MODEL FOR MEGAPROGRAMMING

October 9, 1990

Will Tracz

MD 0210
IBM Federal Sector Division
Owego, N.Y. 13827

OWEGO@IBM.COM
(607) 751-2169

Abstract

"Currently, software is put together one statement at a time. What we need is to put software together one component at a time." — **Barry Boehm, at the Domain Specific Software Architecture (DSSA) Workshop, July 11-12, 1990.**

Megaprogramming, as defined at the first ISTO Software Technology Community Meeting, June 27-29, 1990, by Barry Boehm, director of DARPA/ISTO, is component-based software engineering and life-cycle management. The goal of this paper is to place megaprogramming in perspective with research in other areas of software engineering (i.e., formal methods and rapid prototyping) and to describe the author's experience developing a system to support megaprogramming.

The paper, first, analyzes megaprogramming and its relationship to other DARPA research initiatives (CPS/CPL — Common Prototyping System/Common Prototyping Language , DSSA — Domain Specific Software Architectures, and SWU — Software Understanding). Next, the desirable attributes of megaprogramming software components are identified and a software development model (The 3C Model) and resulting prototype megaprogramming system (LILEANNA — Library Interconnection Language Extended by Annotated Ada) are described.

Keywords: domain modeling, formal methods, inheritance, parameterized programming, rapid prototyping, software engineering, and software reuse.

1.0 Introduction

"Megaprogramming is the type of thing you can go into a 3-star general's office and use to explain what DARPA is going to do for them to make their software less expensive and have better quality." — Barry Boehm, at the ISTO Software Technology Community Meeting, June 27-29, 1990.

Software researchers and developers have long pursued the goal of increased software productivity and quality. As the programming profession matures and basic research into programming languages and formal methods advance, opportunities are emerging to apply some of these results to the software development process. This paper is about component-based programming or *megaprogramming*, a term coined by Barry Boehm[2] at DARPA/ISTO, which is an essential element of the DARPA Software Strategic Plan[1]. Reusing software components, instead of re-writing them, is a long held[16], intuitively appealing, if not obvious, approach to increasing productivity and quality. Systems developed based on reusable software artifacts, in principle, should cost less (partially attributable to a shorter schedule), and contain fewer defects because of the "tried and true" parts used in its composition. Unfortunately, a one-dimensional view of quality as being the "absence of defects" is not sufficient to explain the necessary attributes of software that make it reusable (i.e., portability, flexibility, reliability, useability, and understandability are other essential attributes). The observation that "quality can not be tested into a program, but needs to be designed into a program," is especially applicable to megaprogramming.

The goal of this paper is to examine the technical foundations of megaprogramming and to assess their effectiveness for increasing the interoperability, adaptability, and scaleability of its components (i.e., the quality of its components). To this end, this paper is organized into three sections. The first section summarizes and analyzes the megaprogramming vision initially presented as part of the DARPA Software Technology Plan[21]. The next section introduces a conceptual model for reusable software components (the 3C Model[23]) based on separating a component's context (what can change) from the concept it encapsulates (the interface it exports) and its content or implementation. The final section describes work in progress on a megaprogramming implementation, LILEANNA[24] (Library Interconnection Language Extended by Annotated Ada), which combines the formal methods of ANNA[14] and the parameterized programming capability of OBJ[11]

2.0 Megaprogramming Vision

"Software productivity improvements in the past have been accidental because they allow us to "work faster". DARPA wants people to "work smarter" or to avoid work altogether." — Barry Boehm, at the Domain Specific Software Architecture (DSSA) Workshop, July 11-12, 1990.

Megaprogramming is envisioned as a giant step toward[2] increasing "development productivity, maintenance productivity, reliability, availability, security, portability, interoperability and operational capability[2]." Megaprogramming will incorporate proven, well-defined components whose quality will evolve, in the Darwinian sense. Megaprogramming requires the modification of the traditional software development process to support component-oriented software evolution. Domain-specific software architectures need to be defined and implemented according to software composition principles and open interface specifications. The resulting software assets need to be stored and accessed in a repository ideally built on a persistent object base, with support for heterogeneous software components in distributed environments. Finally, additional environmental capabilities (e.g., hypermedia) are needed to provide software understanding at the component and architectural levels.

The subsections that follow describe some of the focal points of the DARPA Software Technology Plan[21] related to megaprogramming. In particular, an environment to support megaprogramming (Megaprogramming Software Team) and the generation and promotion of megaprogramming components (Megaprogramming Software Interchange) are addressed.

---

[1] Prior to Boehm's use of the term "megaprogramming", Joseph Goguen[11] suggested the term *hyperprogramming* to refer to a similar, if not identical, programming paradigm. The author has suggested using the term *programming-with-the-large*[24] to emphasize the granularity of the objects being manipulated.

[2] The analogy used by Barry Boehm was that, historically speaking, one might view machine language programming as resulting in productivity at a snails pace, assembler language programming — a turtle's pace, programming in FORTRAN, C or Ada — walking, and megaprogramming as walking with seven league boots.

2.1 *Megaprogramming Software Team*

"Configuration = Components + Interfaces + Documentation
Software Team = Configuration + Process + Automation + Control." — Bill Scherlis, at the ISTO Software Technology Community Meeting, June 27-29, 1990.

The goal of the megaprogramming software team is to create an environment to:

1. "manage systems as *configurations* of components, interfaces, specifications, etc.,
2. increase the *scale of units* of software construction (to modules), and
3. increase the *range of scales* of units of software interchange (algorithms to subsystems)[21]."

The key elements of the megaprogramming software team are:

- **Component sources** — currently, components under consideration are from reuse libraries (e.g., SIMTEL20[5] or RAPID[20]) or COTS (Commercial Off-The-Shelf) software (e.g., GRACE[1] or Booch[3] components). Application generator technology is desirable to provide for adaptable modules while re-engineered components (e.g., CAMP[17]) could provide additional resources. It is desirable to move toward new customizable components with a rapid prototyping capability.

- **Interface definitions** — currently, there exists an ad hoc standard consisting of Ada package specifications and informal documentation. It is desirable to develop a Module Interconnect Formalism (MIF) with hidden implementations supported by formal analysis and validation tools.

- **System documentation** — currently, simple hypertext systems are supporting the (often ambiguous and incomplete) textual documentation associated with software components. It is desirable to create a repository-based, hypermedia environment that provides traceability between artifacts and supports the capture, query, and navigation of domain knowledge.

- **Process structure** — currently, there exists no predictable software development process. It is desirable to develop an evolutionary development life cycle with support to domain engineering, integrated requirements acquisition, and reverse/re-engineering.

- **Process Automation** — currently, CASE tools are either stand-alone or federated (e.g., Unix[3]). It is desirable to integrate the tools and create a meta-programming environment to support process description and refinement.

- **Control/Assessment** — currently, only a priori software metrics and process instrumentation exists. It is desirable to integrate the measurement process with tool support and to create a cost-estimation capability.

The megaprogramming software team initially expects to draw resources from the STARS (Software Technology for Adaptable Reliable Systems) SEE (Software Engineering Environment) program. Future tools will be contributed by Arcadia[22], CPS/CPL[6] (Common Prototyping System/Common Prototyping Language), DSSA (Domain Specific Software Architectures)[18], POB (Persistent Object Bases), SWU (Software Understanding), and REE (Re-Engineering) programs. Interface and architecture codification will be supported by a Module Interconnect Formalism (MIF), which is an outgrowth of the CPS/CPL program.

The goal of MIF is to adequately describe a software component such that its selection and use can be accomplished without looking at its implementation. The component interfaces will include, not only the entry points, type definitions and data formats (e.g. Ada package specification), but a description of its functionality, side effects, performance expectations, degree and kind of assurance of consistency between specification and implementation (reliability), and appropriate test cases. DSSA will provide the initial avenue for the application of this technology. (An architecture is a collection of interfaces.) Incremental asset creation and customization will be guided by the CPS prototyping technology.

Asset capture and re-capture will be supported by SWU's design record, hypertext browsing capability, and REE. The design record will provide a "common data structure for system documentation and libraries[21]". The suggested data elements in a design record include:

- code,
- test cases,

---

[3] Unix is a trademark of AT&T Bell Laboratories.

- library and DSSA links,
- design structure,
- access rights,
- configuration and version data,
- hypertext paths,
- metric data,
- requirement specification fragments,
- PDL texts,
- interface and architecture specifications,
- design rationale,
- catalog information, and
- search points.

2.2 Megaprogramming Software Interchange

"Software Interchange = Software Team + Convention + Repository + Exchange." — Bill Scherlis, at the ISTO Software Technology Community Meeting, June 27-29, 1990.

The goal of the megaprogramming software interchange is to "enable wide-area commerce in software components[21]". The megaprogramming software interchange, which is integrated with the megaprogramming software team, consists of the following elements:

- **Conventionalization** — currently, conventions are emerging. It is desirable to create a cooperative decision and consensus mechanism that supports adaptable, multi-configuration libraries, which present a standard search capability.

- **Repository/Inventory**— currently, repositories support code storage only. It is desirable to retain, assess, and validate other software assets such as architectures, test cases, specifications, designs, and design rationales.

- **Exchange/Brokerage** — current intellectual property rights and government acquisition regulations are stifling a software component industry. It is desirable to populate certain application domains (via DSSA) and to support the creation of an electronic software component commerce by defining mechanisms for access control, authentication/certification and establishing composition conventions.

The megaprogramming component interchange expects intially to draw software components from the reuse libraries in STARS and DSSA with future support derived from POB, and CPS/CPL (MIF).

3.0 Conceptual Model for Software Components

"Before components can be reused, there needs to be components to reuse."

As discussed in the previous section, megaprogramming requires the definition of proven, well-defined components that are implemented according to software composition principles. This section presents a formal framework for developing reusable software components that leverage the compositional capabilities of the megaprogramming language LILEANNA (covered in the next section of this paper). A conceptual model[24] is described that distinguishes between three distinct aspects of a software component:

1. the **concept** or abstraction the component represents,
2. the **content** of the component or its implementation, and
3. the **context** that component is defined under, or what is needed to complete the definition of a concept or content within a certain environment.

These three aspects of a software component make the following assumptions about their environment:

1. There is a problem space (application domain) that can be decomposed into a set of concepts (or objects if one prefers using an object-oriented paradigm).
2. There is a solution space that is characterized by the contents (implementations) of the concepts.

3. The solution space is populated by several different implementations, or "* .parameterized⁴" implementations that can be instantiated by different contexts within the solution space.

Before proceeding further into the material in this section, it is important for one to realize the subtle implications that "dynamic binding" has on one's approach to programming. The conceptual model described in this section assumes a programming language and environment with all binding of parameters done prior to run time (with the exception of actual parameters passed to subprogram operations). The model recognizes that binding can occur at or before compile time, and at load/link edit time. This view of binding, to some readers, may appear limiting (which, in some sense, it is), but this limitation, in reality, is a trade-off for early error detection (strong typing), which, in some application areas, is considered to be of greater importance.

The rest of this section defines the terms context, content, and concept, in more detail and describes their relationships to modularization, specification, interface design and parameterization.

3.1 Three Aspects of a Software Component

This conceptual model for software components is motivated by the need to develop **useful, adaptable,** and **reliable** software modules with which to build new applications. These three needs are addressed individually by the model.

1. A **useful** component meets the high-level requirements of at least one concept necessary to design and implement a new software application.
2. An **adaptable** component provides a mechanism such that modules can be easily tailored to the unique requirements of an application.
3. A **reliable** component is one that accurately implements the concept that it defines.

This conceptual model for software components, referred to as the **3-C model,** is based on three aspects of a software component: concept, context, and content. These three terms are addressed individually in the subsections that follow.

3.1.1 Concept

"Domain analysis is the building up of a conceptual framework, informal ideas and relations; the formalization of common concepts." – Ted Biggerstaff, MCC.

The **concept** represented by a reusable software component is an abstract description of "what" the component does. Concepts are identified through requirement analysis or domain modeling as providing the desired functionality for some aspect of a system. A concept is realized by an interface specification and an (optionally formal) description of the semantics (as a minimum, the pre- and post-conditions) associated with each operation. An Ada package specification (operations, type and exception declarations) for a stack abstract data type, with its behavioral semantics described in Anna[14], is an example of a reusable software concept.

3.1.2 Content

"The ability to convert ideas to things is the secret of outward success." – Henry Ward Beecher.

The **content** of a reusable software component is an implementation of the concept, or "how" a component does "what" it is supposed to do. The software component conceptual module assumes that each reusable software component may have several implementations that obey the semantics of it's concept (e.g., operational specifications are the same, but the behavioral specifications are different). The collection of (28) stack packages found among Grady Booch's[3] components is an example of a family of implementations for the same concept (stack).

---

⁴ Perhaps "generalized" is a better word.

3.1.3 Context

"Understanding depends on expectations based on familiarity with previous implementations." — Mary Shaw, SEI.

One of the failures of software reuse is that user's expectations of a reusable software component do not meet the designer's expectations of the reusable software component (the square-peg-in-the-round-hole syndrome). By explicitly defining the context of a reusable software component at the concept and content level, and formally specifying its "domain of applicability", the user can better select and adapt the component for reuse.

The context of a reusable software component takes on three dimensions:

1. the **conceptual context** of a reusable software component — how the interface and semantics of the module relate to the interface and semantics of other modules,
2. the **operational context** of a reusable software component — what the characteristics of the data being manipulated are, and
3. the **implementation context** of a reusable software component — how the module depends on other modules for its implementation.

Parameterization, inheritance and importation of scope through the use of abstract machine interfaces are all language mechanisms that assist in separating context from content. Within the framework of the 3-C model, one uses these language constructs as follows:

1. one specifies the **conceptual context** of a software component by using inheritance to express relationships between concepts (module interfaces). This occurs when two concepts share the same syntax and semantics.
2. one defines the **operational context** of a software component by using genericity to specify data and operations on the data being manipulated by a module (at the conceptual or implementation level).
3. one decides on the **implementation context** of a software component by selecting the operations to be used for and by the implementation of a module. These operations are external to the component. Inheritance or importation of scope are the two languages mechanisms that support the definition of a module's implementation context.

One should note the explicit separation of the roles of code and type inheritance in the model. Type inheritance is used to express the conceptual context of a module. The conceptual context of a software module forms a true partial order in that the concept inheriting another concept "is a" subtype of the latter concept. Code inheritance is used as an implementation mechanism and may or may not be the same as the type inheritance used to express the conceptual context of the concept associated with the software component for which the implementation is being created.

An example of conceptual context is a stack that can be used to describe the interface of a deque (double ended queue). The operational context for a deque is the type of the element being stored. The implementation context of a particular deque implementation might be a sequence abstraction. That is, the implementation would be designed to refer to operations in an abstract machine interface found in a sequence concept, which could have several implementations (e.g., array or linked list). Alternatively, the deque could be indirectly implemented (i.e., generated in the megaprogramming sense) by simply

1. renaming some of the operations in an implementation of the stack (i.e., Push and Pop would become Push_Right and Pop_Right),
2. adding some new operations (Push_Left and Pop_Left), and
3. inheriting the rest (e.g. Print, Length, Is_Empty, etc.).

Using the syntax of LILEANNA, the following megaprogram would generate the (parameterized module) deque described above:

```
make Deque[ Triv ] is
    Stack [ Triv ] * (rename ( Push => Push_Right )
                             ( Pop => Pop_Right )
                             ( Stack => Deque )
                    * ( add Push_Left, Push_Right )
    end;
```

The selection of an implementation, or the content of the concept is determined by trade-offs in context. Clearly, knowing the characteristics of the type of data structure being manipulated will lead to more efficient implementations. This can result in the population of a reuse library with several efficient implementations of the same (parameterized) concept, each tailored to a particular context. At design time, a programmer could identify the concept and define the context it is being manipulated under based on requirements or operating constraints. At implementation time, the programmer could instantiate an implementation of the concept with the conceptual contextual information plus any other contentual contextual information necessary.

Separating context from concept and content complements the work of Parnas[19] in suggesting that the quality of software can be improved by isolating change. It has been demonstrated that software is more reusable, or more easily maintained, if the types of possible modifications to the software are taken into consideration at design time.

4.0   LILEANNA

LILEANNA (LIL Extended with ANNA (Annotated Ada) [14]) is an implementation of LIL (Library Interconnect Language), proposed by Joseph Goguen [9] as a MCL (Module Composition Language) for the programming language Ada[25]. LIL is a language for designing, structuring, composing, and generating software systems. It is based on the work of Goguen and Burstall on the language CLEAR[4] and Goguen on OBJ[8]. LIL was first introduced at the Ada Program Libraries Workshop in Monetary California. It was later refined for publication in IEEE COMPUTER[10]. Since then it has been the interest of several researchers[7, 12, 13, 24].

The primary design goals of LIL were:

1. to make it easier to reuse software written in Ada,
2. to facilitate the composition of Ada packages,
3. to support an object-oriented style of design and documentation for Ada,
4. to rapidly prototype new applications by integrating executable specifications with the controlled manipulation of source code,
5. to avoid recompilation, and
6. to support maintenance of Ada programs and families of programs.

The power of megaprogramming in LILEANNA centers on the ability to compose new packages with package and subprogram expressions via the make statement. Existing packages may be manipulated through package expressions to specify the instantiation, aggregation, renaming, addition, elimination or replacement of operations, types or exceptions.

LILEANNA supports the structuring and composition of software modules from existing modules. One can

1. instantiate a parameterized module to create
    a. implementations of operations,
    b. a simple package/module, or
    c. a parameterized package/module (generic).
2. Compose/structure modules by
    a. combining other modules (inheritance and multiple inheritance) (e.g., merging two module's operations and types),
    b. adding something[5] to an existing (inherited or instantiated) module (e.g., adding an operation),
    c. removing something from the interface of an existing module (e.g., hiding an operation),
    d. renaming something (e.g., purely textual changing the name of operation in an interface),
    e. selecting from a family of implementations, or
    f. replacing something in an existing module (i.e., a pure swap — a remove and add combination).

The result of evaluating a LILEANNA composition/megaprogramming statement (i.e., a make statement) is an executable Ada package specification and body that either is

1. a "stand-alone" flat module (nothing imported), or
2. a hierarchy, with selected functionality imported and perhaps repackaged.

Note that since there is no inheritance in Ada, composition that uses inheritance will need to either import all modules in the inheritance hierarchy (being careful to rename those which might result in ambiguity), or include

---

[5] Where "something" is a sort/type, operation, exception, or in some cases, an axiom.

all necessary functionality directly in the implementation (package body). In either case, the resulting user interface (package specification) should not be cluttered by such details.

4.1 Formal Foundations of LILEANNA

LILEANNA has its formal foundations in category theory[6] and in initial and order-sorted algebras. These concepts form the basis for advances in algebraic specifications and type theory. Many type systems are based on the concept of an algebra. An algebra defines a set of values and the operations on them just as an abstract data type defines the data of the type and provides operations on them.

Program semantics in LILEANNA are expressed in first order predicate calculus rather than using re-write rules (a la OBJ) as a way of implementing conditional order-sorted equational logic.

4.2 LILEANNA Language Constructs and Examples

LILEANNA is a language for formally specifying and generating Ada packages. LILEANNA extends Ada by introducing two entities: theories and views, and enhancing a third, package specifications. A LILEANNA package, with semantics specified either formally or informally, represents a template for actual Ada package specifications. It is used as the common parent for families of implementations and for version control. A theory is a higher level abstraction, a concept (or a context), that describes a module's syntactical and semantic interface. A view is a mapping between types, operations and exceptions.

Programs can be structured/composed using two types of hierarchies:

1. **vertical**: levels of abstraction/stratification, and
2. **horizontal**: aggregation and inheritance (type and code).

LILEANNA supports this with two language mechanisms

1. **needs**: import dependencies, and
2. **import, protect,** or **extend**: three forms of inheritance, and **includes,** a subtyping construct.

Theories are an encapsulation mechanism used to express the requirements on generic module parameters. Theories also play a role in building horizontal and vertical hierarchies by defining the interface requirements for modules that later can be instantiated with a more concrete implementation. Views map theories to theories, or theories to packages, or pieces of packages. One powerful feature of LILEANNA is the encapsulation of parameters in theories. With this capability, the semantics of parameters can be formally specified and the domain of applicability of a module can be explicitly qualified.

The generative capability of the LILEANNA is provided by package expressions, a "super make"[7] feature for creating new packages from existing packages through horizontal, vertical and generic instantiation. Package expressions manipulate Ada packages and their contents based on their relationships to LILEANNA packages, theories and views. The basic operations supported are importation in the form of inheritance, specialization in the form of instantiation, generalization, and aggregation. Finally, the contents of modules can be manipulated through * *package operators* by indicating what entities are being added, hidden, renamed, or replaced.

LILEANNA goes beyond the Ada instantiation capability in that generic packages can be composed to create new generic packages without themselves being instantiated. Partial instantiations are also possible. A view is used to instantiate a generic package. Default views can be computed if only package name is supplied. Alternatively, mappings of formal to actual parameters may form an in-line view as part of a package expression.

The following example illustrates several LILEANNA language constructs. In the example, the package *Integer_Set* is made from a parameterized LILEANNA package, *LIL_Set*. This example is very similar to the instantiation of an Ada generic, except that in Ada, the instantiation process is done at compile time. In LILEANNA, the generic instantiation is done prior to compile time. This results in Ada source code which is ready to be compiled, composed or further instantiated.

---

[6] Goguen has suggested that LILEANNA is based on another 3-C model — Category theory, Colimits, and Comma Categories.

[7] Make is a UNIX term and command for the process of selectively compiling and linking compiled outputs to make an executable module.

```
make Integer_Set is LIL_Set[Integer_View] end;
```

Attention should be paid to the view (shown below), *Integer_View* (from theory *Triv* to the Ada package *Standard*), used in the make statement above. There is an explicit mapping between the type *Element* and the type *Integer*. The point to be emphasized is that this mapping can be given a name and reused in other instantiations.

```
view Integer_View :: Triv => Standard is
    types (Element => Integer);
end;
```

Alternatively, as shown below, the instantiation could have been stated as

```
make Integer_Set is
    LIL_Set [ view Triv => Standard is types (Element => Integer); ]
end;
```

In this case, the view does not have a name, but the mapping is explict to this particular instantiation.

The following example illustrates the use of horizontal and vertical composition. A generic package (*Short_Stack*) is generated by selecting an array implementation (*List_Array*) of the list interface theory (*List_Theory*) needed by the LILEANNA package (*LIL_Stack*). It is assumed that the LILEANNA package (*LIL_Stack*) has a comparable Ada package (*Stack*) and that an explicit view may or may not exist between them.

```
make Short_Stack is
    LIL_Stack   -- inherit Stack Package (horizontal composition)
    needs (List_Theory => List_Array)
                -- supply array package (vertical composition)
end;
```

The following is an example of a make statement that instantiates the generic LILEANNA package *Sort* according to the view *Nat_Default* (not shown), which maps the Natural numbers and the pre-defined linear order relationship onto the theory of partially ordered sets.

```
make Sort_Lists_of_Naturals is
    Sort[Nat_Default]
    needs (ListP => Linked_List)
end;
```

An example of a more involved make statement using multiple inheritance and package operators follows. It is based on an existing set of Ada packages that defines an Ada-Logic Interface[15] package for reasoning.

```
make New_Ada_Logic_Interface is
    Identifier_Package +
    Clause_Package*(hide Copy) +
    Substitution_Package +
    DataBase_Package +
    Query_Package*(add function Query_Fail (C: Clause;
                                            L: List_Of_Clauses)
                                   return Boolean)
            *(rename ( Query_Answer => Query_Results ))
end;
```

The result is a merged package specification where,

1. the *Copy* operation is not available on *Clauses*,
2. an additional operation, *Query_Fail*, now augments those inherited from the specification, *Query_Package*,
3. the *Query_Answer* operation is not available in the resulting interface, instead, the *Query_Results* operation can be invoked.

5.0 Conclusion

"We should stand on each others shoulders, not on each others feet." — Peter Wegner[26]

Megaprogramming is a new programming paradigm that requires both a critical mass of software components and a disciplined approach to program design and specification. This paper has presented one approach to megaprogramming that is based on a formal model (the 3-C Model) for developing reusable software components. This model gives insight into the relationships between type inheritance, code inheritance, and parameterization that is essential for providing the adaptability and interoperability of software components. The corresponding implementation, LILEANNA, serves as a valuable vehicle for exploring megaprogramming concepts.

6.0 References

1.   Berard, E.V. Creating Reusable Ada Software. *Proceedings of the National Conference on Software Reusability and Maintainability*, September 1986.

2.   Boehm, B. DARPA Software Strategic Plan. *Proceedings of ISTO Software Technology Community Meeting*, June 27-29 1990.

3.   Booch, G. *Software Components with Ada.* Benjamin Cummings, 1988.

4.   Burstall, and Goguen, J.A. The Semantics of CLEAR, a Specification Language. *Proceedings of the 1979 Copenhagen Winter School of Abstract Software Specification*, pages 292-332, 1980.

5.   Conn, R. The Ada Software Repository. *Proceedings of COMPCON87*, February 1987.

6.   Gabriel, R.P. (editor). Draft Report on Requirements for a Common Prototyping System. *in ACM SIGPLAN Notices*, 24(3):93-165, March 1989.

7.   Gautier, R.J. A Language for Describing Ada Software Components. *Proceedings of Ada-Europe Conference*, May 26-28 1987.

8.   Goguen, J.A. Some Design Principles and Theory of OBJ-0, a Language for Expressing and Executing Algebraic Specification of Programs. *Proceedings of Mathematical Studies of Information Processing*, pages 425-473, 1979.

9.   Goguen, J.A. LIL - A Library Interconnect Language. *in Report on Program Libraries Workshop, SRI International.*, pages 12-51, October 1983.

10.   Goguen, J.A.   Reusing and Interconnecting Software Components. *IEEE Computer*, 19(2):16-28, February 1986.

11.   Goguen, J.A.   Hyperprogramming: A Formal Approach to Software Environments. *Proceedings of Symposium on Formal Approaches to Software Envrionment Technology*, Joint System Development Corporation, Tokyo, Japan, January 1990.

12.   Harrison, G.C.   An Automated Method for Referencing Ada Reusable Code Using LIL. *Proceedings of Fifth National Conference on Ada Technology and Fourth Washington Ada Symposium*, March 17-19 1987.

13.   Liu, D.B.   A Knowledge Structure of a Reusable Software Component in LIL.. *Proceedings of Sixth National Conference on Ada Technology*, March 14-17 1988.

14.   Luckham, D. and vonHenke, F.W.   An Overview of Anna, A Specification Language for Ada. *IEEE Software*, 1(2):9-22, March 1985.

15.   Madhav, N. and Mann, W.   Abstract Specification of Automated Reasoning Tools: An Ada-Logic Interface, Program Analysis and Verification Group, Stanford University, 1989.

16.   McIlroy, M.D.   Mass Produced Software Components. *Proceedings of NATO Conference on Software Engineering, edited by Naur, P., Randell, B. and Buxton, J.N.*, pages 88-98, 1969.

17.   McNicholl, D.G., Palmer, C., et al.   Common Ada Missile Packages (CAMP) Volume I: Overview and Commonality Study Results, McDonnel Douglas Astronautics Company, :AFATL-TR-85-93, May 1986.

18.   Mettala, E.G.   Domain Specific Software Architectures presentation at ISTO Software Technology Community Meeting, 1990.

19.   Parnas, D.L.   A Technique for Software Module Specification with Examples. *Communications of the ACM*, 15(5):330-336, May 1972.

20.   Ruegsegger, T.   Making Reuse Pay: The SIDPERS-3 RAPID Center. *IEEE Communications Magazine*, 26(8):816-819, August 1988.

21.   Scherlis, W.L.   DARPA Software Technology Plan. *Proceedings of ISTO Software Technology Community Meeting*, June 27-29 1990.

22.   Taylor, R., et al .   Foundations for the Arcadia Environment Architecture. *Proceedings of Third Symposium on Software Development Environments*, pages 1-13, November 1988.

23.   Tracz, W.   The Three Cons of Software Reuse. *Proceedings of Fourth Workshop on Software Reuse Tools.*, 1990.

24.   Tracz, W.J.   *Formal Specification of Parameterized Programs in LILEANNA*, PhD thesis, Stanford University, 1990. In progress.

25.   U.S. Department of Defense, US Government Printing Office, The Ada Programming Language Reference Manual, 1983.

26.   Wegner, P.   Varieties of Reusability. *Proceedings of ITT Workshop on Reusability Programming*, September 1983.

Ada Net

John McBride

Planned Solutions

AdaNET

Presented to
RICIS '90 Software Engineering Symposium

November 8, 1990

Presented by
John McBride
Planned Solutions, Inc.

AdaNET Program

- Five Year R & D Effort to Advance the State of Software Engineering Practice

- National Facility in West Virginia to Increase U.S. Productivity, Economic Growth & Competitiveness

- Enhance Existing AdaNET System to Provide a Life Cycle Repository for Software Engineering Products, Processes, Interface Standards, & Related Information Services

Purpose and Scope

- Transfer Software Engineering Technology Within the Federal Sector & to the Private Sector

    - Reusable Software Components Useful in All Phases of Lifecycle

    - Engineering Process Descriptions for Developing Adaptable & Reliable Systems & Software Worthy of Reuse

    - Interface Standards

        - More Consistency in System Features,
        - Simpler System Integration,
        - Aid in the Use of Metrics as Quality Predictors

    - Related Information & Services

        - Software Engineering Help Desk
        - Conference Listings
        - References
        - Networking to Other Databases
        - E Mail

AdaNET Goals

- Establish a National Center for the Collection of Software Engineering Information
- Provide On-Line Life Cycle Repository
- Promote a Cultural Change Necessary to Improved Quality & Efficiency
- Provide a Platform for Research in Technology Transfer

Planned Solutions, Inc.

AdaNET Benefits

- Decrease Software Costs
- Improve Quality of Software Systems

Planned Solutions, Inc

AdaNET is a National Resource
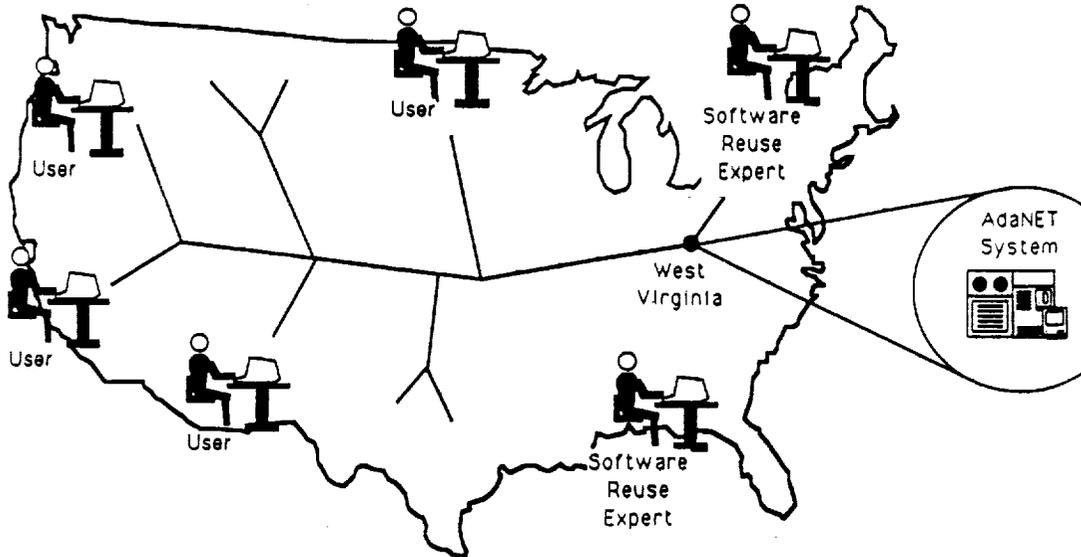
Accessible Via InterNET and TeleNET Public Access Dial Up

Users of AdaNET

Small Companies - Reusable Components and Software Engineering Help Desk will Allow These Companies to be More Competitive

Large Companies - Large, Complex Systems can be Built More Reliably and at Lower Cost with Reusable Components

Academia - Facilitates Teaching and Research in Software Engineering With Reusability

U. S. Government - Spinback Benefits to Government Software Developers

Major Research and Technology Issues

Application and Dissemination Policies	Software Reuse Strategies		AdaNET Architecture
• Interagency Agreements • Customer Licenses • Data Rights • Title and Use Guarantees • Liability • Organization Type • Charges and Profits • International Clients • Military Restrictions • • •	• Domain • Type • Granularity • Selection • Configuration • • •	• Modification • Classification • Retrieval • Assistance • Qualification • • •	AdaNET Context • Operating Modes • Security and Integrity • User Interface AdaNET Services to Access Resources AdaNET Resources • Information • Products • Experts

Planned
Solutions, Inc

AdaNET Enhancements

AdaNET Service Version Two (ASV2) Current System

- Hosted on Data General
- CEO Office Automation Product Organized Files In Drawers and Folders
- Keyword and Textual Search

ASV3 (late 1991)

- Unix Based
- Integrate JSC/Barrios Developed Autolib & Army/RAPID Derived Technologies
- Natural Language Query, Facets, Keyword Search

ASV4 (late 1994)

- Object Management Support for Full Life Cycle Traceability

Planned
Solutions, Inc

AdaNET User Registration

Mountain NET

P.O. Box 370

Dellslow, W.V. 26531

(304) 296-1458

(304) 296-6892 FAX

1-800-444-1458 help desk (Peggy Lacey)

Current AdaNET Products and Services

Reusable Software		Publications	
Army Ada Software Repository	(227)*	• Citations	(678)
STARS Repository	(In process)	• Newsletters	(19)
NASA/JPL Components	(In process)	• Standards	(92)
Products		Conferences	
• Services	(40)**	• Announcements	(112)
• Software	(141)	• Paper Calls	(20)
E-Mail		News	
		• Abstracts	(129)
		• User Contributions	(21)
Training		Contracts	
• Guided Study	(102)	• Awards	(161)
• Self Study	(21)	• RFPs	(177)

* - Functional Areas
** - Unique Files

Summary

- Life Cycle Approach to Reuse Can Provide a Significant Impact on Software Productivity

- Software Engineering Information Provides Knowledge Transfer

- AdaNET is an Operational Program with a Prototype Development and Evaluation Cycle

Planned Solutions, Inc

POSIX and Ada Integration
in the
Space Station Freedom Program

Robert A. Brown
The Charles Stark Draper Laboratory, Inc.

Overview

- POSIX Overview

- POSIX Execution Model

- Ada Execution Model

- SSFP Flight Software Ada Requirements

- POSIX/Ada Integration

POSIX Overview

- Portable Operating System Interface
  for Computer Environments

- IEEE sponsored standards development effort
  - Voluntary participation
  - Concensus standard (75% required for approval)

- Purpose
  - Define standard OS interface and environment
  - Based on UNIX
  - Support application portability at source code level

- Family of open system standards

POSIX Working Groups

- P1003.0: Guide to POSIX Open Systems Environment

- P1003.1: System Interface

- P1003.2: Shell & Tools

- P1003.3: Testing & Verification

- P1003.4: Realtime

- P1003.5: Ada Language Bindings

- P1003.6: Security Extensions

- P1003.7: System Administration

- P1003.8: Networking

- P1003.9: Fortran Language Bindings

- P1003.10: Supercomputing

- P1003.11: Transaction Processing

POSIX Execution Model
P1003.1

- POSIX process
  - Address space
  - Single thread of control executing in address space
  - Required system resources

- Process management
  - Process creation -- fork() and exec()
  - Process group and session
  - Process termination -- exit(), abort()

- Process synchronization
  - Signals -- sigsuspend(), pause()
  - Wait for child termination -- wait(), waitpid()

- Process delay
  - alarm() and sleep()

POSIX Execution Model
Realtime Extensions

- Priority scheduling

- Binary semaphores

- Shared memory

- Message queues

- Asynchronous event notification

- Clocks and timers
  - High resolution sleep
  - Per-process timers

Ada Execution Model
Language Definition

- Ada program
  - Single address space
  - Multiple threads of control
  - Required system resources

- Task management
  - Task creation -- elaboration, allocator evaluation
  - Organization -- task master
  - Task termination -- normal completion, exception

- Task synchronization
  - Rendezvous

- Task delay
  - Ada delay statement

SSFP Flight Software Requirements

- Multiple real-time programs sharing same processor

- Fixed priority, preemptive scheduler

- Single level dispatcher

- Non-blocking i/o and system calls

- Ability to schedule tasks for periodic execution

- Ability to schedule tasks to respond to specific events

Ada Execution Model
Realtime Extensions

- Scheduling
  - CIFO cyclic scheduler

- Binary semaphores

- Shared data template

- Precision time services

- Event notification
  - CIFO event management

POSIX/Ada Integration
The Problem

- POSIX looks from program outward
  - Semantics defined for processes only
  - Single thread assumption

- Ada looks from program inward
  - Semantics defined for tasks within a program only
  - Single program assumption

- Integration of POSIX and Ada
  - Extend POSIX semantics to multi-threaded processes
  - Extend Ada semantics to multiple programs

POSIX/Ada Integration
A Solution

- Extension of POSIX semantics to multiple threads
  - Define system interface for threads
  - Redefine existing services for multiple threads
    - Signals
    - Fork() and exec()
    - Per process static data
    - Semaphores, events and timers

- Extension of Ada semantics to multiple programs
  - Global task scheduling
  - Definition of shared package semantics
  - Ada interfaces to multiprogramming services
    - Process control -- start, stop
    - Interprocess communication

Session 4
Software Engineering: Issues for Ada's Future

Chair: **Rod L. Bown** , *University of Houston-Clear Lake*

Assessment of Formal Methods for Trustworthy Computer Systems

Susan Gerhart
Microelectronics and Computer Technology Corp. (MCC)

An Assessment of Formal Methods for Trustworthy Systems

Susan Gerhart

MCC Formal Methods / Software Technology

gerhart@mcc.com   512-338-3492

- What are Formal Methods?
- Standards for Trustworthy Systems
- Assessment of FM via SafeIt

C-2

"Applied Mathematics of Software Engineering"
college sophomore through Ph.D. level

Use

logic, set and sequence notation,
finite state machines, other formalisms

Induction arguments, Symbolic reasoning

In

- system models
- specifications
- designs and implementations

For

- highly reliable, secure, safe systems
- more effective production methods
- software engineering education

*NSA Comp. Sec.
U.K. MoD
Tek, ATCS
SEI*

In levels of use

guidance: structuring what to say
rigorous, formal:
generated and worked proof obligations
mechanized: using proof assistants

European

U.S.

A NonExecutable Spec Language: ASLAN

- State-transition based

- First order logic with equality

- Sections

  » Types (builtin and user constructed)

  » Constants & Variables

  » Definitions & Axioms

  » Initial Condition

  » Invariant

  » Constraint

  » Transitions (pre/post conditions)

- Generates verification conditions

  » IC => INV

  » For each t , INV' & PRE'(t) & POST(t) => INV & CON

- Limited type checking

- PASCAL-like syntax

- Levels (of refinement)

  » Additional VCs

- Derived from Ina Jo research (R. Kemmerer at UCSB)

Portion of an ASLAN Spec

state {
TYPE ...

        book is structure of (
                title : string,
                author : string,
                subject : string),
        copy,
        copies is set of copy
VARIABLE ...
        db: library,
        staff: users,
        borrower(copy): user,
        next_id: pos_int

Properties {
INITIAL

        db = empty & staff = empty & next_id = 1
INVARIANT
        forall c:copy
        (c isin db -> available(c) xor borrower(c)~=noone)
        &
        cardinality(db,next_id-1)

Steps
TRANSITION check_out(c:copy, u:user, s:user)
ENTRY   c isin db & available(c) & s isin staff &
        under_lim(u)
EXIT   borrower(c) becomes u

...

An ASLAN-generated Verification Condition

consistency conjecture for check_out(c:copy, u:user, s:user):

Invariant before

(forall c:copy
c isin db' -> c[available] xor c[borrower] ~= noone
&
c isin db' & c[available] & s isin staff' & under_lim'(u)
&

Transition Effects

~c[available] & c[borrower]=u
&
db = db'
&
staff = staff')

⊃ ->

Invariant After

(forall c:copy
c isin db -> c[available] xor c[borrower] ~= noone
&
true)

e, indicating that no back-
ocess has been selected for exe-
o interrupts are active, the pro-
ille, and the Select operation
n spontaneously. It is specified

—

g = none

and' = background
: ready
' ∈ ready
:ndler' = θIntHandler

—

ran a part of the interface be-
kernel and an application, Se-
ternal operation of the kernel
appen whenever its precondi-
. The precondition is

none ∧ ready ≠ ∅

ssor must be idle, and at least
round process must be ready to
rst part of this precondition is
icitly, and the second part is im-
: predicate

ready

value of current is selected
but the specification does not
: choice is made — it is non-
ic. This nondeterminism lets
cation say exactly what pro-
may rely on the kernel to do:
guarantee that processes will
d in a particular order.
nondeterminism is a natural
e of the abstract view I have
specification. Although the
at implements this specifica-
ministic — if started with the
esses in a certain state, it will
the same process — it ap-
ondeterministic if you pay a
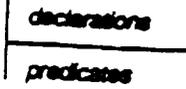to the set of processes that are
re done in the specification.
e kernel selects the new cur-
the specification says that it
Because of the static schedul-
ich determines that after the

In Z, the schema X is defined by the form

```
X
declarations
predicates
```
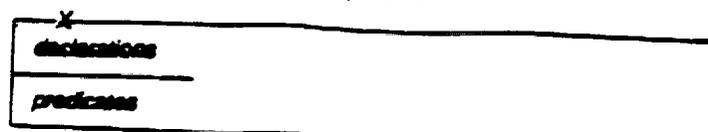
Global functions and constants are defined by the form

```
declarations
predicates
```

The declaration gives the type of the function or constant, while the predicate gives its value. Here, I define only the Z symbols used in this article:

Sets:

$S: \mathbb{P}\, X$	S is declared as a set of X's.
$x \in S$	x is a member of S.
$x \notin S$	x is not a member of S.
$S \subseteq T$	S is a subset of T: Every member of S is also in T.
$S \cup T$	The union of S and T: It contains every member of S or T or both.
$S \cap T$	The intersection of S and T: It contains every member of both S and T.
$S \setminus T$	The difference of S and T: It contains every member of S except those also in T.
$\emptyset$	Empty set: It contains no members.
$\{x\}$	Singleton set: It contains just x.
$\mathbb{N}$	The set of natural numbers 0, 1, 2, ...
$S: \mathbb{F}\, X$	S is declared as a finite set of X's.
$max(S)$	The maximum of the nonempty set of numbers S.

Functions:

$f: X \rightarrowtail Y$	f is declared as a partial injection from X to Y (described in the handler definition on p. 23).
dom f	The domain of f: the set of values x for which $f(x)$ is defined.
ran f	The range of f: the set of values taken by $f(x)$ as x varies over the domain of f.
$f \oplus \{x \mapsto y\}$	A function that agrees with f except that x is mapped to y.
$\{x\} \lhd f$	A function like f, except that x is removed from its domain.

Logic:

$P \wedge Q$	P and Q: It is true if both P and Q are true.
$P \Rightarrow Q$	P implies Q: It is true if either Q is true or P is false.
$\theta S' = \theta S$	No components of schema S change in an operation.

Z Structure - schema

```
Stop
ΔState
────────────
running ∈ background

background' = background \ {current}
ready' = ready \ {current}
current' = none
θIntHandler' = θIntHandler
```

For this operation to be permissible, the processor must be running a background process. This process is removed from background and ready, and the current
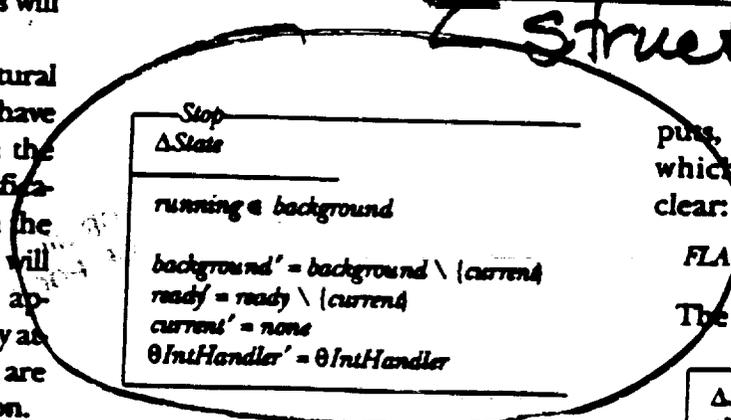
puts, the process identifier and a flag, which takes one of the values set or clear:

FLAG ::= set | clear

The SetReady operation is:

```
SetReady
ΔState
p?: PID
flag?: FLAG
────────────
p? ∈ background

flag? = set ⇒ ready' = ready ∪ {p?}
```

CRUISE-ACT

CRUISE-MON

PEDAL-OVERRIDE and
not BRAKE-ON

REACH-SP

ACTIVATE

CRUISING

SELECT-SP

sp(SELECT-SP)

MONITOR

PEDAL   SPEED

PEDAL-MON   SPEED-MON

STOP-TEST/SCHED-PED   STOP-MAIN/SCHED-MAIN

throughout PEDAL-MON - TEST-PED-DEF
throughout SPEED-MON - MAINTAIN-SP
throughout SELECT-SP SELECT-SP
throughout CRUISE-MON - TEST-PED-DEF and CHECK-SP

STOP-TEST - stopped(TEST-PED-DEF)
STOP-MAIN - stopped(MAINTAIN-SP)
SCHED-PED - schedule)(start!(TEST-PED-DEF),n seconds)
SCHED-MAIN - schedule)(start!(MAINTAIN-SP),n seconds)

STATEMATE,
i-Logix, Inc

Figure 4: Cruise State Zoom-in

Tools Catalogue

Languages

- NonExecutable:

  Z, VDM (at least 2 flavors), ASLAN, Larch, Estelle, ...

- Executable: (prototyping)

  Miranda, OBJ, me too, StateChart, Caliban,D, Prolog

Static Analysis

FUZZ, ASLAN + (all executable systems)

Language-tailored Environments

Raise, Larch, Gist, Statemate

Concurrency-centered

CSP, CCS, Unity, Petri-nets, Spec, Lotos, ...

Temporally focused

L.0, ASLAN-RT, RTL, Timed CSP, Tempura, TempLog,

Theorem Provers

Boyer-Moore, HOL, Clio, m-EVES, B, Isabelle, OBJ,
EHDM, Gypsy, uRAL...

Sample Applications in Progress

Project	Parties	Problem	Status
CICS	Oxford PRG IBM Hursley	Transaction Processing	Released, Measured (??)
Cleanroom	IBM FSD NASA SEL	Embedded, Restructurer	Released Evaluated
ZEE	Tektronix	Oscilloscopes	On-going
Avalon/C++	C-MU	Atomicity	Preliminary
GKS, OA Doc.	British Standards Institute	Graphical, Documents	Published
Hypertext Ref. Model	Dexter Group Denmark	Hypertext Concepts	Report VDM90
SXL	GTE Labs	Protocols	In use
L.0	Bellcore	Protocols	In use
CASE	Praxis	Object Manager	Report, product
Anti-MacEnroe Device	Sydney Inst. Technology	Tennis Line Fault Detector	Report (Occam,CSP)
Security	Honeywell Ford Aero. Digital TIS	LOCK Multi-net Gateway Secure VMS Trusted Mach	In progress " " "
VIPER	RSRE, Cambridge	Microprocessor Tools	Reports Newsletter
Verified Stack	CLinc	Microp, assembler, O.S.	Reports
Oncology	U. Wash.	Cyclotron	Starting
Reactor Control	Parnas, Ontario Hydro	Shutdown Certification	Reports, Certified
Murphy	U.C. Irvine	Safety	Reports
SACEM	French RR	Train Control	ICSE12

Trust ↓

Standards

Security "Orange Book" -NSA

Safety

    MoD 0055/56 (interim)
      Hazard analysis +
      Safety-critical development Process

    SafeIT - goals   (UK DTI)
- technically sound
- generic
      sector - transportation/medical ...
      application
- feasible
- international

[NIST standards]

Motivation

    Safe systems
    High integrity industry competition
    Trade advantage (1992)

SOFT NEWS

Software safety focus of new British standard

Galen Gruman, Soft News Editor

The British Defence Ministry expects to issue a a new software-safety standard this spring that will require the use of formal methods and mathematical verification on all safety-critical software. Only developers who prove that their software is not safety-critical will be exempt from the requirements.

The standard, MoD-Std-0055, will ban the use of assembly language, limit the use of high-level languages like Ada to safe subsets, and require the use of static analysis. It also sets standards for project engineers. It will require that an engineer sign off on the software's safety compliance, that the engineer have taken accredited formal-methods instruction within the past two years, and that an independent engineer with similar accreditation also sign off on the system. This is similar to the responsibility and requirements enforced on systems-safety engineers for the overall project.

The 0055 standard will be in effect for two years, during which time the Defence Ministry will revise it on the basis of industry's experience. The intent is to develop a long-term standard, said Kevin Geary, a software consultant for the British navy's procurement department who is working on the 0055 standard. The ministry is also working on MoD-Std-0056, a hazard-analysis standard that will help software developers determine where to apply formal methods and mathematical verification, Geary said. "Both mathematical verification and hazard analysis must be performed to provide software with acceptable risk. Neither is adequate alone," said Nancy Leveson, a software-safety expert and a computer-science professor at the University of California at Irvine.

Pros of formal methods. The 0055 standard has been called a "landmark" by those in the software-safety and formal-methods communities, who argue that assigning responsibility to software engi-

neers, as has been tradition in hardware engineering, will help encourage changes in development methods that will help assure safe systems. Safety is increasingly important because software is becoming a greater part of critical systems like aircraft controls, medical devices, nuclear-power plants, early-warning defense systems, and missile controls, they said.

Most software-engineering standards depend on testing, which is not always reliable, Geary said. "The problem with software is that you must test against specifications. If you didn't get the specifications right, you might not get the software right," he said. However, mathe-

---

The forthcoming UK Defence Ministry standard will require the use of formal methods and mathematical verification for safety-critical software.

---

matical analysis of formal specifications notations can be used to find errors in the specifications, Leveson said.

The increasing number of tools like Zed, Vienna Development Method, Spade, and Malpas will help make the implementation of formal methods possible because these tools can perform static analyses of information flow and semantics quickly, rather than in the years required with manual techniques, Geary said.

Formal methods and mathematical verification are often considered too difficult to apply, Geary conceded. "There is a lot of unease, but it's quite surprising that there are a lot of key people who've

come around after looking at it," he said. Geary cited IBM's British development center, which decided for commercial reasons — not for government or other outside requirements — to use the Zed formal method on CICS development. "People's resistance is based on ignorance," Geary said.

Another source of resistance is the confusion between formal, mathematical methods and mathematical correctness. "Correctness is a meaningless goal for real systems. For example, do you have a 'correct' airplane?" Leveson said. "A more realistic and useful goal is to build a system that satisfies a given set of functional and mission requirements while at the same time trying to satisfy constraints of safety, security, and cost," she said. Many of these goals involve trade-offs in setting priorities, she said.

Leveson compared formal methods to traditional hardware engineering: "Engineers build formal mathematical models and then use analysis methods to determine whether the model has certain desired properties," she said, "which should be the role of formal methods in software engineering." (Leveson's "Safety as a Software Quality" essay in this issue's QualityTime, on pp. 88-89, gives more details about this process.)

"Both software engineers and hardware engineers specify design," Geary said. "The only difference is how tangible [the product] is," he said.

Still, software engineers do face a burden that their hardware counterparts generally do not: the complexity of their product, said Martyn Thomas, chairman of Praxis Systems, a software-engineering consulting firm in Bath, England, that does much work in safety engineering. Traditional engineers like bridge-builders "never had techniques for design, which is more important for software because that's where the complexity comes in. It's not a software problem but a design-complexity problem," he said. Whether overtly or covertly, the profes-

Intro

"17. Specification
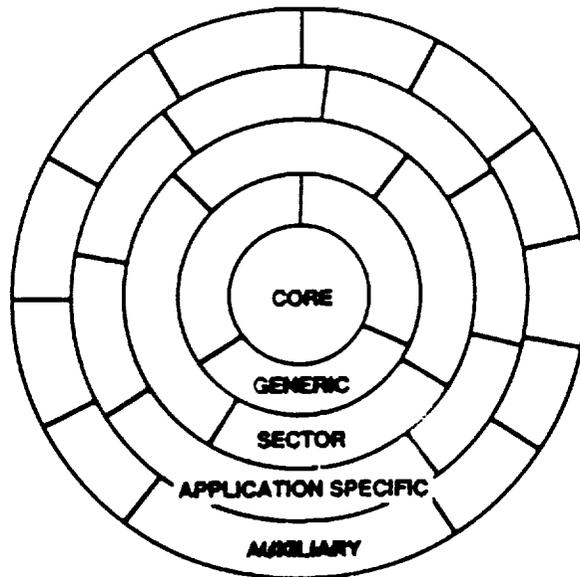
17.1   Safety Critical Software shall be specified using formal mathematical techniques. A specification of the Safety Critical Software shall also be produced in clear English. Both specifications shall be included as part of the Procurement Specification. A list of formal mathematical specification techniques is given in Annex L."

- VDM, Z, OBJ, HOL, CCS, CSP, Temporal Logic, Lotos

Figure 1   Structure of the Framework

Components

Safe IT

CORE

GENERIC

SECTOR

APPLICATION SPECIFIC

AUXILIARY

Hierarchy

Core Standards
- Principles
- Terms and Concepts
- Methods for Standards
  Development

GENERIC
STANDARDS

SECTOR
STANDARDS

APPLICATION SPECIFIC
STANDARDS

PART 1 :   GENERAL
REQUIREMENTS

PART 2 :   SYSTEM
SAFETY ELEMENTS

PART 3 :   SAFETY INTEGRITY
REQUIREMENTS

PART M

PART N

Manufacturing

Transport

Process
eg
Petro-
Chemical

Medical

- Interlocking
- Moulding Machines

- Emergency Shut
  Down Systems
- Boilers

Auxiliary Standards
- Fault Trees
- FMEA
- Q/A
- V & V

Table 1: Summary of Objectives
Overall Objective: Assured Integrity

Main Objectives	Sub-Objectives
adequate specification of safety features *Adequat Spec*	clarity and precision
	management of complexity
	self consistency
validity *Validity*	valid translation of PES specification to software
	defined and valid specification of other PES components
	defined and valid specification of external systems eg physical, software, human and maintenance systems
	fault detection, toleration and management defined
implementation (code) satisfies specification *Satisfies Spec*	clarity and precision
	management of complexity
	self consistency
	adequate refinement
integrity of management and development process *Process Integrity*	commitment of senior management
	motivated and competent staff
	active and effective management controls
integrity maintained during operation *Integrity maintained*	maintenance specified during design
	integrity of maintenance process
	integrity of modifications
	security of software code
assurance *Assurance*	comprehension
	empirical and analytical evidence
	recognition of residual doubt and fallibility
	demonstration to second and third parties
	valid reasoning system

Objective: adequate specification		
Sub-Objectives	Techniques	IEC techniques
clarity and precision	formal specification language with defined syntax and semantics; graphical representation; application specific language, engineering notations — block diagrams, Process and Instrumentation diagrams, algebra, z transforms, discrete equations; natural language annotations; structured natural language; subsets of languages	formal mathematical modelling; data flow diagrams; finite state machines/state transition diagrams; structure diagrams
management of complexity	abstraction; modularity; information hiding; structured design technique	formal mathematical modelling; data flow diagrams; finite state machines/state transition diagrams; structure diagrams
self consistency of specification	animation — proof of invariants and theories; semantics for notations; review and inspection; execution of properties — prototyping of selected properties; testing	prototyping/animation; simulation; functional testing; formal mathematical modelling; Fagan inspections; formal design reviews
validity	see next table	

Formal Spec. Lang./Methods

ASLAN - state transition

Z - set-based

Larch - theories

CCS, UNITY - concurrency

Statechar - finite state

Tools

Provers ← Symbolic analysis → Tests

graphics + formal text + informal text

Usable, ,

Sub-Objectives	Techniques	IEC techniques
clarity and precision	formal specification language with defined syntax and semantics; graphical representation; application specific language; engineering notations — block diagrams, Process and Instrumentation diagrams, algebra, z transforms, discrete equations; natural language annotations; structured natural language; subsets of languages	formal mathematical modelling; data flow diagrams; finite state machines/state transition diagrams; structure diagrams
management of complexity	abstraction; modularity; information hiding; structured design technique	formal mathematical modelling; data flow diagrams; finite state machines/state transition diagrams; structure diagrams
self consistency of specification	animation — proof of invariants and theories; semantics for notations; review and inspection; execution of properties — prototyping of selected properties; testing	prototyping/animation; simulation; functional testing; formal mathematical modelling; Fagan inspections; formal design reviews
adequate refinement	logical reasoning; review/inspection; testing; static analysis; experimentation; experience in the field; diversity of tools and people; use of subset of programming language; languages that can cope with different levels of abstraction	Fagan inspections; formal design reviews; formal proof of program; sneak circuit analysis; walkthroughs; functional testing

Review
Prove

Spec₁
⋮
Spec₂
⋮
Code

Refinement, Transformation, Implementation

Test

— Lang. Dfn.

K.4 Integrity of process

106   As in any engineering endeavour, the integrity of the development and management process is essential to the achievement and assurance of integrity. There is a requirement that the system is what it seems, that documentation is adequate and under configuration control and that the claims made about the system are valid.

Objective: integrity of process		
Sub-Objectives	Techniques	IEC techniques
active and effective management controls	QMS to ISO 9000; independent QA; automated configuration management; manual configuration management; clear delineation of authority and responsibility for safety; adequate project planning, cost estimation and monitoring tools and procedures	checklists; Fagan inspections; formal design reviews
commitment of senior management to safety and quality	awareness campaigns; certification approval schemes; demonstration of economic benefits; regulatory inspection; liability; standards; safety culture	
motivated and competent staff	competency of key staff (eg to BCS Safety Critical Curricula); experience in application domain and of software techniques used in project; qualification to Chartered Engineer status; status and pay; professional development; certification; safety culture	

107   Note: Within this technical framework only recommendations concerning management controls and competency of staff can be made. Other factors are important and should be addressed during the project (eg safety culture considered in the selection of contractors). Similarly, broad security issues have not been considered. It may be possible in future versions of the Framework to reference out these objectives to a QMS standard.

(i) Maintenance and modification activities are inadequate. It should be appreciated that maintenance can be a dominant source of common mode failures in redundant systems. Also, maintenance will be particularly important in long lifetime systems or systems which are expected to evolve.

(ii) Security of the embedded code is violated. General consideration of security are outside the scope of this framework, for further discussion see the publications from the DTI Commercial Security Centre [9].

(iii) Failures in the system violate the stated conditions under which the integrity is ensured. The detection, toleration and management of such changes are addressed in the section on validity (K.2) and are not considered further in this section.

109   The need for maintenance of the hardware and software will affect the design of the software structure and fault handling, reporting and recovery mechanisms. This is addressed in section K.2.

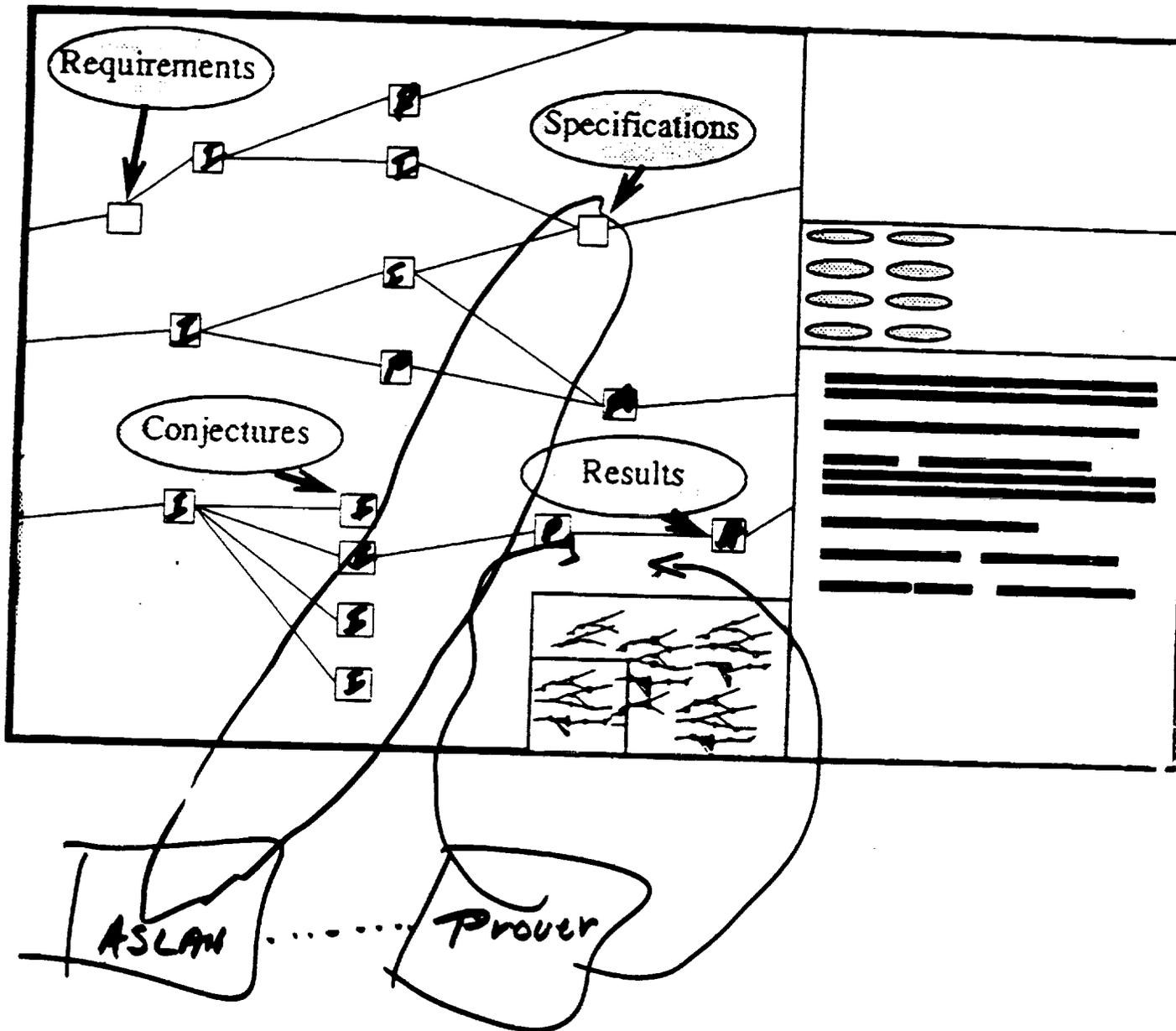| Objective: integrity of software maintained during operation		
Sub-Objectives	Techniques	IEC techniques
integrity of maintenance process	maintenance planning and standards; manual configuration management; automated configuration management; authorisation procedures; availability of qualified staff; development facilities; Quality Management Systems	
integrity of modifications	application of design standards and development standards to modifications; regression testing; procedures for assessing impact and importance of change; modularity and structuring	
security: software code unchanged	robust storage media; security; administrative access controls; passwords; safety critical data not changed by operational staff; encryption and other fault tolerant techniques	error correcting codes

comprehension	timely provision of documentation; visible lifecycle; satisfaction of other framework objectives	
empirical and analytic evidence	*See 'satisfaction of specification'. In addition require:* proof deliverable; appropriate V&V techniques — dynamic testing, logical reasoning, documented reviews; evaluation of operating experience of identical and similar systems; use of proven or certificated components	formal proof of program; checklists; Fagan inspections; formal design reviews; boundary value analysis; error guessing; error seeding; performance modelling; simulation; test coverage; functional testing
recognition of residual doubt	claim limits; design guidance (e.g. 'no single failure criterion') on system level diversity	
recognition of fallibility	diversity of tools, techniques, people and organisations — IV&V, ISA; diverse proof checker; diversity of other tools; robust design — fault detection and containment; QA and technical review	checklists; Fagan inspections; formal design reviews; fault detection and diagnosis
demonstration to second or third parties	involvement of customer; QA within a QMS; liason with customer QMS; compliance with Health and Safety at Work Act and other relevant legislation and standards; safety record log or accomplishment summary; certification of people, procedures and components	checklists; Fagan inspections; formal design reviews
valid system of reasoning	accepted mathematical inference system or calculus; empirical evidence; common language	formal mathematical modelling

Mcc SpecTra Screen Mock-up

See: RICIS hypermedia conf., Dec. GERM

```
label:: books
type:: declaration
date:: Jun 14 10:05 1990
author:: greene
Contents:: books is set of book
```

Figure 7 *Contents of the* Decl *node labeled* books

Besides the one-of links (denoting the set membership relation), there are is-of-type and depends-upon links (*v* is-of-type *t* when *v* is a state variable and *t* is its type and Decl *d1* depends-on Decl *d2* when the declaration *d2* mentions the formal entity declared in *d1*). These links are by default invisible (to cut down on the clutter) but can be displayed at the user's request. For example, a user can click on a transition node (a node containing the entry and exit conditions of an ASLAN transition) and ask for all of the nodes in the specification on which this transition depends. SpecTra then highlights all of the nodes in the specification which can be reached by starting at the clicked upon node and following depends-upon links. Thus the graphical representation of an ASLAN specification is easier to browse than the textual representation. SpecTra is also able to highlight all the nodes which depend upon a user specified node. This eases the task of specification modification as users can be pointed to all the parts of the specification which will be affected by a change.

Figure 8 *Informal requirements linked to formal specification*

Using these new node and links types, formal ASLAN specifications can be entered and browsed within Germ. Additionally, I/P/A structured informal requirements may coexist in the database and these informal notions may be linked to the portion of the formal specification which is their formalization. For example, in the process of coming up with requirements for the library database, the following issue arose. Should the concepts *book* and *copy* be identified? Arguments (pro and con) were given and it was decided that these two notions should be distinguished. The position taken was that a book was something abstract and that a copy was an instance of that abstraction. The links between this posi-

MCC Extensions: Animation

Animation of
Process (threads) spec

Figure 2 Relationship of the risk and safety integrity levels to the Safety Lifecycle Model
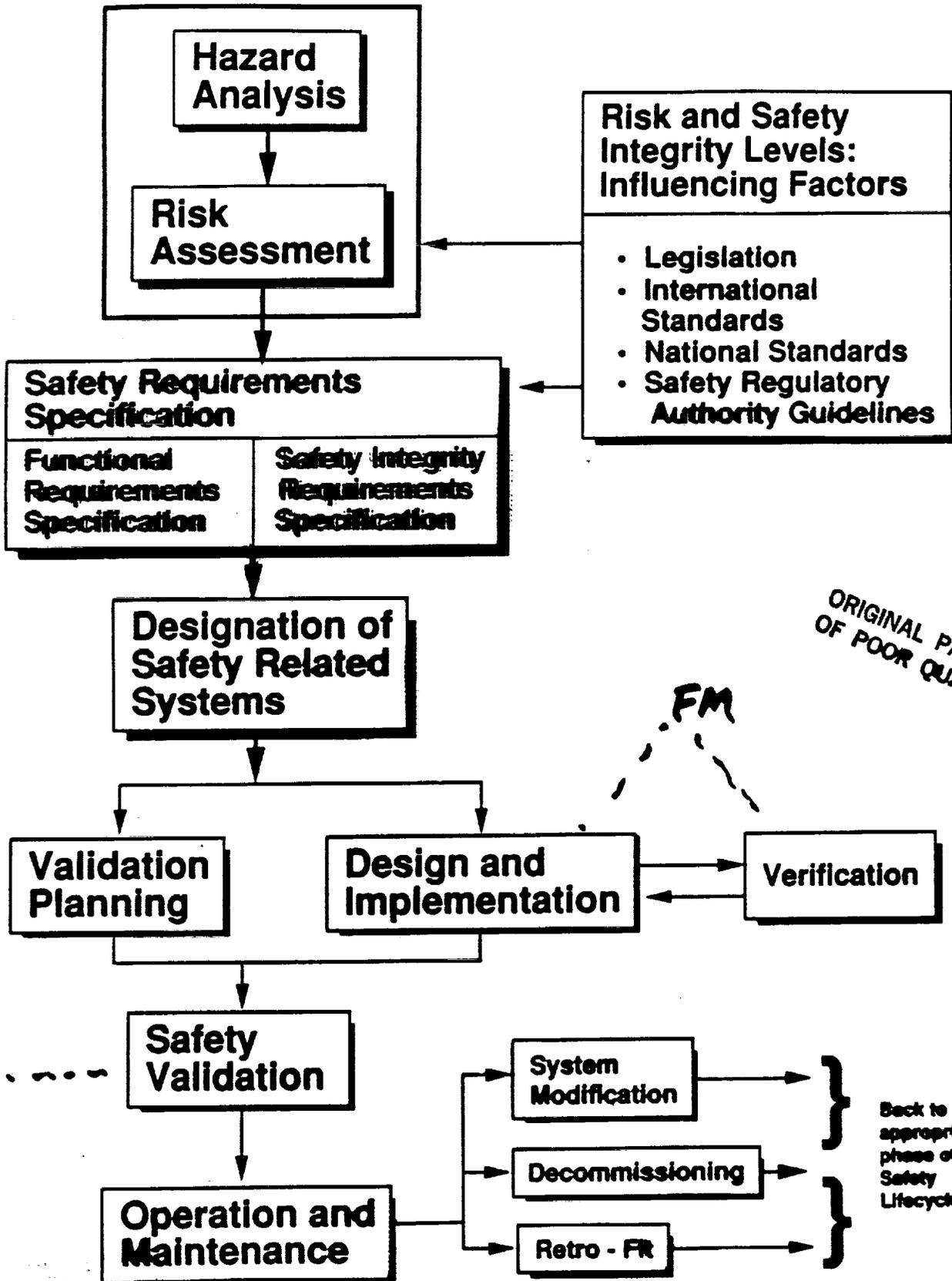
CONCLUSIONS

SafeIT could be used to define
support needed for trustworthy
system development, e.g.
Space Station

Techniques X Sector, X Ada
Application X ≡

Formal Math.
Safety Eng.
Software Eng.
. . . .

Preliminary Assessment — FM
- Evidence for effectiveness
     IEEE Sw, Computer, TSE Sept. '90

     FM 89 ··· Springer Verlag 1991

- Education basis
     SEI MSE, texts, network groups

- Tool environments weak
     Proven - - - ? - - -> Formal CASE
     Interfaces

MCC Formal Methods Project

1) Transition Study

    Survey, assess
    Experiments
    Education

    14 organizations, incl. NASA, MITRE,
                      Rockwell

2) SpecTra

    Hypertext platform
        Nodes - specs.
        Links - process, dependencies

    "Executable specs"
        Logic & functional prog.

    Hybrid methods
    Integrating tools

MCC

Formal Methods Transition Study

Interest is growing worldwide in the application of precise mathematical techniques to the specification and design of hardware and software systems. In fact, European successes in this area, commonly called *Formal Methods*, have already led governments to require that the techniques be used for safety critical systems.

MCC's Software Technology Program proposes a one-year in-depth study of Formal Methods techniques and the tools that support them. Drawing upon significant research experience at MCC, we will assess the state of the art worldwide and determine the implications for a variety of North American industries.

This proposal describes the background, rationale, and contents of the funded study, including its timeline and deliverables. Our goal is to provide executives with the information they need to ascertain their own companies' requirements in the Formal Methods area. For those whose interest calls for further technology development, this study will also establish a plan for appropriate research and development work.

Background, Rationale: Formal Methods, a body of techniques supported by powerful reasoning tools, offer rigorous and effective ways to model, design, and analyze systems. Several research groups, primarily in Europe, have generated specification, implementation, and verification techniques for a broad class of systems, and have cast the techniques into industrially usable forms. Their affiliated companies have already employed several of these techniques in the development of real-world hardware and software applications. Attention by governments and industry is increasing as well, due in large part to a growing concern with the high risks of faulty computer control in systems critical to life and property. Indeed, certain combinations of Formal Methods are now seen as necessary for ensuring that these systems meet existing regulations and standards, or that they avoid legal liability repercussions. And there are other, broader applications for these techniques as well; in particular, they can help circumvent many of the expensive problems of general software development practices, such as late discovery of errors and poor communication among end users, designers, specifiers, and implementors.

MCC is in a unique position to build on the progress in Formal Methods. Even today, a number of tools and techniques developed in MCC research laboratories can be brought to bear. For example, Software's issue-based design methodology can be integrated with Advanced Computing Technology's declarative language technology and with externally developed Formal Methods-based toolsets. MCC researchers have proposed several novel ways in which to exploit MCC-developed techniques to advance Formal Methods research. Moreover, researchers in the Software Technology and Computer-aided Design programs are investigating CoDesign—design and analysis techniques spanning both hardware and software. So that we may capitalize on worthwhile outside developments as they occur, MCC's International Liaison Office closely monitors the maturation of Formal Methods techniques in Europe and gauges industrial and government interest in both Europe and the U.S. At the same time, MCC's experiences with technology transfer continue to give us bountiful insights into the problems and operations of MCC's sponsoring organizations.

Content of Study: We propose to study Formal Methods issues as they directly relate to North American companies. First, we will determine how Formal Methods can help these companies meet demands for higher quality, possibly regulated software-intensive systems. Second, we will pinpoint how the companies can exploit Formal Methods in current environments for more productive software development processes.

The study will explore the issues and topics that pertain to a full-scale Formal Methods research effort at MCC, including:

Fundamental concepts of Formal Methods—what is a formal method, and how does it work?

Training and instructional material—sample course outlines, evaluation of course offerings.

Modes of using formal methods—specification, verification, documentation, refinement; integration with object-oriented and other widespread approaches; consistency of artifacts from requirements through code.

Survey of major applications—summaries of Formal Methods projects to date, interpretations of collected project data, evaluation of successes and failures, derived guidelines for applications.

Tools survey—catalog of editors, syntactic/semantic checkers, theorem provers, and other tools; MCC experiments with North American and European toolsets; assessment of state of toolsets.

Models of formal-based software development—injection of techniques into standard productivity, risk, and QA models; scenarios of future development processes.

Regulatory and legal trends in safety and security—the high-integrity market sector; research funding patterns (U.S., Europe, and Japan); forecasts of error and development costs, adoption patterns, optimistic and pessimistic scenarios.

Transitional tips—what to teach, to whom, and follow-through; projects to try; pitfalls, motivation, and so on.

Experimental results—results of using MCC technology and personnel, along with imported tools, instructors, consultants, and other studies, to apply Formal Methods to industrially relevant problems. These experiments will illustrate many of the above topics.

Research needs and strategy.

<u>Timeline and Deliverables</u>: The proposed study will be conducted from September 1, 1990, to September 30, 1991. At the end of this period, participants will receive a comprehensive report covering the topics outlined above, together with video overviews, tool demonstrations, and thorough accounts of experimental protocols and results. Drafts of the report's topics will be available at quarterly intervals; midterm and final reviews and information sessions will occur at the MCC site; and at least one formal inter-

action will be designed according to the specific interests of each participant (within the domain expertise limits of MCC personnel).

The study in its entirety will be proprietary to participants for one year, after which MCC may distribute it more widely. Selected sections reporting experimental results and new insights of interest to the research community may be published as technical reports and papers during the course of the study, both to further the field and to establish the MCC Formal Methods initiative in the research community.

<u>Costs</u>: Costs for the study will be targeted to ten participants at $60,000 each. Membership is open to all MCC shareholders and associates; non-member companies can opt to participate in MCC for the one-year study period only, paying a special Project Associate fee of $7,500 in addition to the study participation fee. Should there be more than ten participants, additional personnel will be added to increase the study's scope and depth.

A full-scale, multiple-year Formal Methods initiative will be proposed in mid-1991. While the study's report will motivate many of the initiative's activities, it will not constitute a full definition of those activities. Study participants have no commitment beyond September 1, 1991; however, if a participant does elect membership in the initiative, it may deduct $25,000 from the cost of membership over the first two years.

<u>Personnel</u>: The MCC researchers who will conduct the study are broadly experienced in the theory and application of Formal Methods techniques and tools. They are also experts in tracking and forecasting technology trends. The study coordinator, Dr. Susan Gerhart, has led a major U.S. formal verification project and participates in international Formal Methods strategic activities. Other project members are experts in a variety of tools (already assembled at MCC), techniques, and theories and have applied them to industrially interesting problems. This unique group has been cooperating for a year and will be complemented by consulting expertise from outside MCC as well as from related MCC projects.

Issues Related to Ada 9X
John McHugh
Computational Logic, Inc.

Recent Ada 9X
Activities

John McHugh

Baldwin / McHugh Associates
Durham, North Carolina

8 November 1990

Ada 9X Activities ▬▬▬▬▬▬▬▬▬▬▬▬▬▬

OVERVIEW

- **Ada 9X**

- **The 9X process**

- **Issues for Critical Systems**

Ada 9X Activities ▬▬▬▬▬▬▬▬▬▬▬▬

Ada 9X

ISO Standards such as Ada must be
reviewed for possible revision every 10
years. The review process can

- Leave the standard unchanged
- Withdraw the standard
- Initiate a revision process

Ada 83 is undergoing a revision. The new
language will be known as Ada 9X.

- The current expected value for X is 3.

The Ada 9X Process

The Ada 9X process is being managed by
the Air Force out of Eglin AFB, Fla. The
project manager is Christine Anderson.

- Revision requests submitted 88-89
- Requirements workshops 89-90
- Distilled to revision issues by IDA
- Requirements document - drafts fall 90
- Inputs still coming from interest groups
- Mapping contractor (Intermetrics) will map
  requirements into revised language

My Subjective View of Process

The following represent my own , distinctly minority view of the process.

- The ground rule that calls for upward compatibility at all costs does more harm than good as it guarantees a more complex language.
- As Ada tries to be all things to all people, dialects and subsets will become necessary.
- A rational approach is probably not possible. Without it, Ada 9X will not be a substantial improvement over Ada 83 and Ada will eventually collapse under its own weight,

Ada 9X Activities

Ada 9 X and Critical Systems

As a part of the revision that Ada is undergoing , the trusted systems community has raised a number of issues. They are summarized in the following slides.

Ada 9X Activities

Requirement A

IDENTIFY AND *JUSTIFY ALL* ELEMENTS OF THE STANDARD THAT PERMIT UNPREDICTABLE PROGRAM BEHAVIOR.

e.g., Program blockage

Integer (1.5) $\overset{?}{=}$ Integer(1.5)

INTENT IS TO ELIMINATE WHERE POSSIBLE AND FORCE ANALYSIS AND COST BENEFIT DECISION ELSEWHERE.

REQUIREMENT A -continued

1) Eliminate most erroneous cases
2) Eliminate "incorrect order dependency"--define order-dependent semantics
3) Define undesirable implementation dependency (UID)
4) UID has defined effect, not cause for "program error"
5) Implementations shall attempt to detect remaining erroneous and UID cases
6) Specific cases of undefined variables:

   a. Majority - URG position on LHS usage

   b. Minority - catch all usage

REQUIREMENT B

EXPOSE IMPLEMENTATION CHOICES

1) Language choices (LRM alternatives)
2) Implementation strategy (storage management, scheduling, etc.)
   - Static choices
   - Dynamic choices
   - What can user control?
   - How can information be shared with others? With tools?

Choices include:

a) Parameter passage
b) Optimization
c) Heap vs stack vs ...storage management

REQUIREMENT C

ALLOW USERS TO CONTROL

IMPLEMENTATION TECHNIQUES

Certain implementation choices lead to explosive growth in possible execution behaviors.

Implementations must honor—or reject with warnings—user directives for items such as parameter passing mechanisms, orders of evaluations, etc.

This is analogous to the representation specification for data.

REQUIREMENT D

IMPLEMENTATIONS SHALL ATTEMPT COMPILE OR RUNTIME ANALYSIS FOR KNOWABLE INSTANCES OF UNSOUND PROGRAMMING AND ISSUE WARNINGS/EXCEPTIONS AS APPROPRIATE.

- Aliasing
- Unsynchronized sharing
- Uninitialized variables
- Etc.

REQUIREMENT E

PROGRAM BEHAVIOR TO BE DEFINED OR PREDICTABLE IN THE FACE OF OPTIMIZATION

We call for further study on the following

- Canonical order of evaluation vs radical optimizations
- Exceptions
- Side effects
- Possibility of pragma control

Page 6

REQUIREMENT F

FORMAL STATIC SEMANTICS AS PART OF

ADA 9X STANDARD

The formal definition to be accompanied by tools that facilitate use for answering questions about the legality and meaning of programs.

While this does not necessarily change the language, development of the definition and tools may contribute to language changes.

N.B. Parameterize formal definition for implementation decisions and architecture/environment.

Ada 9X Activities

REQUIREMENT G

DYNAMIC SEMANTICS AS ONGOING EFFORT WITH AIM OF INCORPORATIONS IN NEXT STANDARD.

This area has enough uncertainty to keep it off the Ada 9X critical path. On the other hand, development of portions of the dynamic semantics as part of the Ada 9X effort should aid in evaluating and understanding proposed language changes.

N.B. Parameterize formal definition for implementation decisions and architecture/environment.

Ada 9X Activities

REQUIREMENT H

ASSERTIONS

MAJORITY
1) Need dynamic semantics for assertions to be useful for proof
2) Suitable form not known
   - Extend Ada expressions
   - Ada vs spec functions
   - Etc.
   ∴ Wait, but work on issue

MINORITY
1) Anna exists
2) Anna is better than nothing
   ∴ Use Anna for now

DON'T PRECLUDE LATER
CHOICE/DECISION

Mixed Results

- Requirements A, B, and D are largely reflected in the Requirements Document
- Requirements C and H have been largely ignored.
- Requirement E has resulted in special consideration being given to the critical systems community.
- Requirements F and G have been completely rejected, but ...

Page 8

Language Precision Team

PRDA issued by Ada 9X project last
spring.

- Supports Ada 9X mapping team
  by providing formal analysis of
  selected language topics

- "Creeping formalism" approach to
  demonstrating utility of formal
  methodology

- May have some influence on Ada 9X
  language

A team led by ORA was issued a contract
during the last days of FY 89.

Ada 9X Activities

Research Issues and Efforts

The language precision team will work with
Intermetrics to model specific aspects of the Ada
language where the application of formal
techniques appears to have promise. These
include optimization and tasking. While the project
is probably worth while, the approach may be less
than satisfactory for a number of reasons.

Ada 9X Activities

Features Interact

In isolation, most Ada features are innocuous. It is in combination that they cause problems. The LPT approach risks ignoring the interactions

- Overloading
- Separate Compilation
- Private types
- Signals and handlers
- Tasking
- Optimization and code generation

Ada 9X Activities

Consider Optimization

Optimization and code generation are difficult to separate. One man's optimization strategy is another's code generation paradigm.

- Ada has no explicit low level parallelism. Most modern architectures do, even if it is *only* a pipeline or a coprocessor.

- Array and vector processors have primitives that are of a *higher* level than the Ada primitives that they implement.

- The ability of the programmer to explicitly handle exceptions from predefined operations makes visible implementation details that are better hidden.

Ada 9X Activities

Reconsider Optimization

The interaction of exception handling, global data, and separate compilation with low level parallelism makes code generation difficult.

- Reordering exception raising operations *can* create unexpected program states or even turn a legal program into an *erroneous* one.
- If the exception is unhandled, this may not matter.
- If the exception is handled in another compilation, the dependencies are difficult to track.
- Without global analysis, the wrong choices are sure to be made sometimes.

Ada 9X Activities

Meanwhile back at Intermetrics

The first Ada 9X Mapping Issues document produced by Intermetrics addresses no issues that are of specific interest to the critical systems community. The issues addressed include:

- Type extensions and polymorphism
- Pointers to static objects
- Changes in visibility rules for operators
- etc.

Ada 9X Activities

What lies Ahead?

The process will inexorably wend its way towards a revised Ada. While some of the warts of the present language may be removed in the process, it is certain that others will spring up to take their place.

The process is under the control of those with a certain vested interest in the status quo.

What is lacking is a long term, radical view of what ought to be. If Ada 9X, like Ada 83 fails to serve the needs of portions of the community, where can they go? What alternatives do they have?

Ada 9X Activities