NASA Conference Publication 10176

# Third NASA Langley Formal Methods Workshop

*Compiled by*
*C. Michael Holloway*
*Langley Research Center • Hampton, Virginia*

Proceedings of a workshop sponsored by the
National Aeronautics and Space Administration,
Washington, D.C., and held at
Langley Research Center, Hampton, Virginia
May 10–12, 1995

June 1995

# TABLE OF CONTENTS

# INTRODUCTION

On May 10-12, 1995, the formal methods team at the NASA Langley Research Center sponsored their third workshop[1] on the application of formal methods to the design and verification of life-critical systems. This workshop brought together formal methods researchers, industry engineers, and academicians to discuss the potential of NASA-sponsored formal methods and to investigate new opportunities for applying these methods to industry problems.

This publication constitutes the proceedings of the workshop. It contains copies of the material presented at the workshop, summaries of many of the presentations, a complete list of attendees, and a detailed summary of the Langley formal methods program. Much of the material contained herein is available electronically through the World-Wide Web via the following URL: http://atb-www.larc.nasa.gov/WS95/proceedings.html.

On behalf of the formal methods team, we thank all of the presenters and attendees for their contributions to making this workshop a great success. We look forward to seeing all of you again at our next workshop.

C. Michael Holloway, *Workshop Co-chairman*
Ricky W. Butler, *Workshop Co-chairman*

[1] Previous workshops were held in 1990 (see NASA Conference Publication 10052) and 1992 (see NASA Conference Publication 10110).

# Final Agenda

PRECEDING PAGE BLANK NOT FILMED

PAGE 4 INTENTIONALLY BLANK

# *Third NASA Langley*

# *Formal Methods Workshop*

formal methods

NASA Langley

***May 10-12, 1995***
***H. J. E. Reid Conference Center***
***NASA Langley Research Center***
***Hampton, Virginia***

***Sponsored by***
***Formal Methods Team***
***Assessment Technology Branch***
***Information & Electromagnetic Technology Division***
***Research and Technology Group***

Proceedings available on the World-WideWeb at
http://atb-www.larc.nasa.gov/WS95/proceedings.html

## *Wednesday, May 10, 1995*

| | |
|---|---|
| 8:00 – 8:45 | **Late Registration** |

### *Session 1: Introduction to Formal Methods*

Chaired by Ricky Butler

| | |
|---|---|
| 8:45 – 9:00 | **Welcome**<br>*Milton Holt, Chief, Information & Electromagnetic Technology Division* |
| 9:00 – 9:30 | **Rationale for Formal Methods**<br>*Ricky Butler, NASA Langley Research Center* |
| 9:30 – 10:30 | **An Informal Introduction to Formal Methods**<br>*Michael Holloway, Ricky Butler, and Paul Miner*<br>*NASA Langley Research Center* |
| 10:30 – 11:00 | Break |
| 11:00 – 12:00 | **An Informal Introduction to Formal Methods (continued)** |
| 12:00 – 12:30 | **Overview of NASA Langley's Formal Methods Program**<br>*Ricky Butler, NASA Langley Research Center* |
| 12:30 – 1:30 | Lunch in NASA Cafeteria |

### *Session 2: LaRC-sponsored Industrial Applications*

Chaired by Ricky Butler

| | |
|---|---|
| 1:30 – 2:00 | **The AAMP5/AAMP-FV Project**<br>*Steve Miller, Rockwell-Collins*<br>*Mandayam Srivas, SRI International* |
| 2:00 – 2:30 | **Union Switch & Signal Project**<br>*Joe Profeta, Union Switch & Signal*<br>*Doug Hoover, Odyssey Research Associates* |
| 2:30 – 3:00 | **Honeywell Software Development Project**<br>*Lance Sherry, Honeywell*<br>*Doug Hoover, Odyssey Research Associates* |
| 3:00 – 3:30 | Break |

### *Session 3: Industry Perspectives on Formal Methods*

Chaired by Michael Holloway

| | |
|---|---|
| 3:30 – 5:00 | **Panel Session**<br>*Scott French, Loral ATC*<br>*Steve Miller, Rockwell-Collins*<br>*Joe Profeta, Union Switch & Signal*<br>*Lance Sherry, Honeywell* |
| 5:00 – 6:00 | **Cash Bar Social in Reid Conference Center** |

## *Thursday, May 11, 1995*

### *Session 4: Software Systems (1)*

Chaired by Ricky Butler

8:30 - 9:30 **Formal Verification for Fault-Tolerant Architectures/PVS Design**
*John Rushby, SRI International*

9:30 - 10:30 **Formal Methods Demonstration Project for Space Applications**
*Ben DiVito, VÍGYAN, Inc.*
*John Kelly, Jet Propulsion Laboratory*

10:30 - 10:45 Break

### *Session 5: Software Systems (2)*

Chaired by Michael Holloway

10:45 - 11:45 **Ada 9X Language Precision Team**
*David Guaspari, Odyssey Research Associates*

11:45 - 12:30 **Introduction to Penelope and Its Applications**
*David Guaspari, Odyssey Research Associates*

12:30 - 1:30 Lunch in NASA Cafeteria

### *Session 6: Hardware Systems*

Chaired by Paul Miner

1:30 - 2:00 **The Formal Verification Technology Used on AAMP5**
*Mandayam Srivas, SRI International*

2:00 - 2:30 **Specification and Verification of VHDL Designs**
*Damir Jamsek, Odyssey Research Associates*

2:30 - 3:15 **Derivational Reasoning System**
*Bhaskar Bose, Derivation Systems Inc.*

3:15 - 3:30 Break

### *Session 7: Researcher Perspectives on Formal Methods*

Chaired by Jim Caldwell

3:30 - 5:00 **Panel Session**
*David Dill, Stanford University*
*Doug Hoover, Odyssey Research Associates*
*Steve Johnson, Indiana University*
*Natarajan Shankar, SRI International*

6:30 - 8:00 **Dinner at Captain George's Seafood Restaurant**

## *Friday, May 12, 1995*

### *Session 8: Research Issues (1)*

Chaired by Michael Holloway

| | |
|---|---|
| **8:30 - 9:15** | **Safety Analysis**<br>*John Knight, University of Virginia* |
| **9:15 - 10:00** | **Non-standard Analysis and Software**<br>*Richard Platek, Odyssey Research Associates* |
| **10:00 - 10:30** | Break |

### *Session 9: Research Issues (2)*

Chaired by Victor Carreño

| | |
|---|---|
| **10:30 - 11:15** | **Hybrid Fault Algorithms**<br>*Pat Lincoln, SRI International* |
| **11:15 - 12:00** | **Model Checking**<br>*David Dill, Stanford University* |
| **12:00 - 1:30** | Lunch in NASA Cafeteria |

### *Session 10: Research Issues (3)*

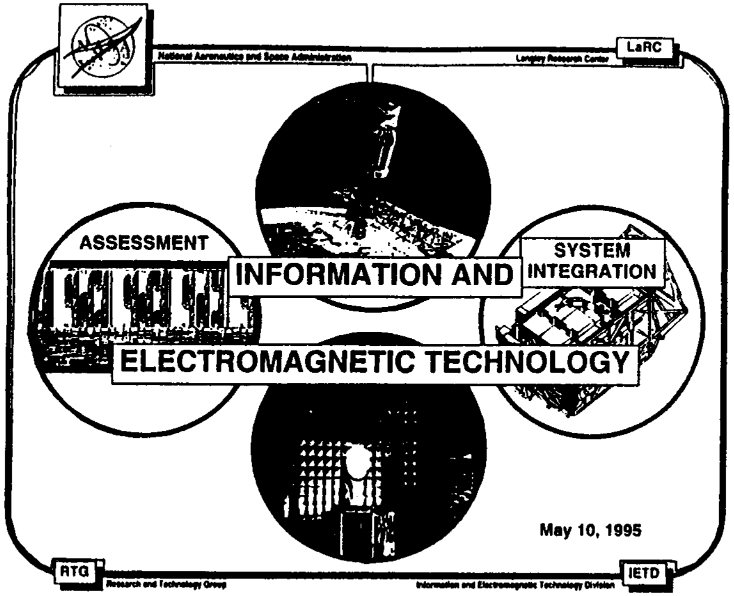
Chaired by Ricky Butler

| | |
|---|---|
| **1:30 - 2:00** | **The DDD Scheme Machine**<br>*Steve Johnson, Indiana University* |
| **2:00 - 2:30** | **Formal Development of a Clock Synchronization Circuit**<br>*Paul Miner, NASA Langley Research Center* |
| **2:30 - 2:40** | **Closing Remarks**<br>*Ricky Butler, NASA Langley Research Center* |

# Session 1: Introduction to Formal Methods

## Ricky W. Butler, Chair

- **Welcome**, by *H. Milton Holt*, Chief, Information & Electromagnetic Technology Division
- **Rationale for Formal Methods**, by *Ricky W. Butler*
- **An Informal Introduction to Formal Methods**, by *C. Michael Holloway*, *Paul S. Miner*, and *Ricky W. Butler*
- **Overview of NASA Langley's Formal Methods Program**, by *Ricky W. Butler*



National Aeronautics and Space Administration

Langley Research Center

LaRC

ASSESSMENT

SYSTEM INTEGRATION

INFORMATION AND

ELECTROMAGNETIC TECHNOLOGY

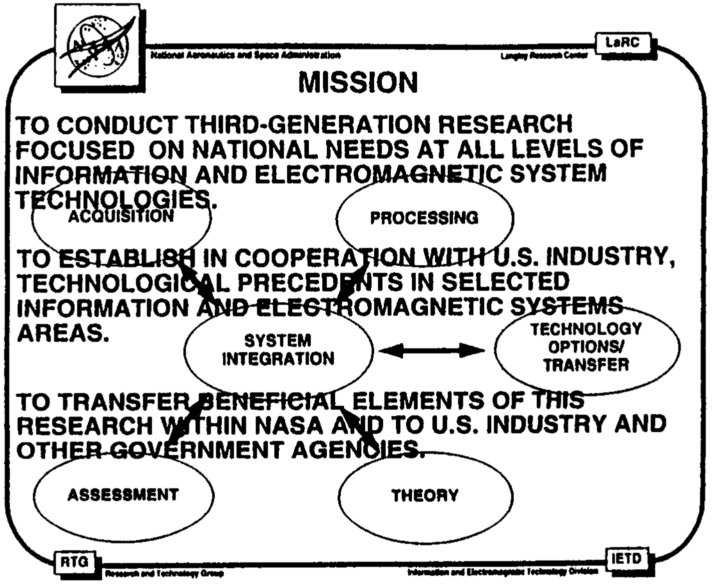
May 10, 1995

RTG

Research and Technology Group

IETD

Information and Electromagnetic Technology Division



National Aeronautics and Space Administration

Langley Research Center

LaRC

## MISSION

TO CONDUCT THIRD-GENERATION RESEARCH FOCUSED ON NATIONAL NEEDS AT ALL LEVELS OF INFORMATION AND ELECTROMAGNETIC SYSTEM TECHNOLOGIES.

TO ESTABLISH IN COOPERATION WITH U.S. INDUSTRY, TECHNOLOGICAL PRECEDENTS IN SELECTED INFORMATION AND ELECTROMAGNETIC SYSTEMS AREAS.

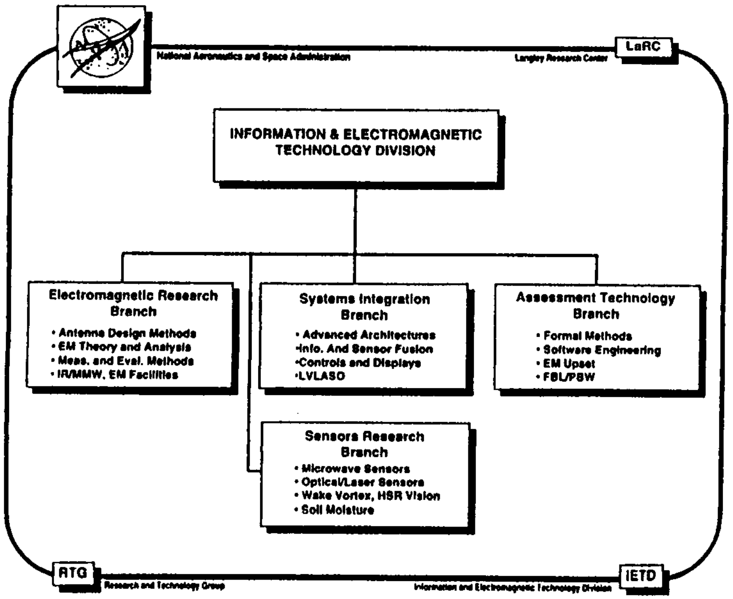
TO TRANSFER BENEFICIAL ELEMENTS OF THIS RESEARCH WITHIN NASA AND TO U.S. INDUSTRY AND OTHER GOVERNMENT AGENCIES.

RTG

Research and Technology Group

IETD

Information and Electromagnetic Technology Division



National Aeronautics and Space Administration

Langley Research Center

LaRC

INFORMATION & ELECTROMAGNETIC TECHNOLOGY DIVISION

Electromagnetic Research Branch
- Antenna Design Methods
- EM Theory and Analysis
- Meas. and Eval. Methods
- IR/MMW, EM Facilities

Systems Integration Branch
- Advanced Architectures
- Info. And Sensor Fusion
- Controls and Displays
- LVLASO

Assessment Technology Branch
- Formal Methods
- Software Engineering
- EM Upset
- FBL/PBW

Sensors Research Branch
- Microwave Sensors
- Optical/Laser Sensors
- Wake Vortex, HSR Vision
- Soil Moisture

RTG

Research and Technology Group

IETD

Information and Electromagnetic Technology Division

PAGE 12 INTENTIONALLY BLANK

# Rationale for Formal Methods

by

Ricky W. Butler

NASA Langley Research Center

May 10, 1995

## Outline

1. The Problem
2. Why Formal Methods Is Needed
   (i.e. Why Standard Practice Won't Work)
3. What is Formal Methods

## A Small Sample of Notorious Design Errors

- Saturation of AT&T network on 15 January 1990 caused by a timing problem in a fault-recovery mechanism.
- Failure to launch STS-1 (synchronization of backup computer)
- Therac 25 medical radiation-treatment machine responsible for the death of two people.
- Loss of data from Voyager at Jupiter (loss of clock sync)
- Patriot failure at Dharan (clock drift)
- In flight tests of the X31, the control system went into a reversionary mode four times in the first nine flights usually due to a disagreement between the two air data sources.
- AFTI-F16—Asynchronous operation, skew, and sensor noise led each channel to declare others failed in flight test 44. Flown home on a single channel. Other potentially disastrous bugs detected in flight tests 15 and 36.
- X29 bug detected by simulation after 162 "at-risk" flights. Analysis showed that the bug could have led to instability and consequent loss-of-aircraft.
- Pentium floating point division bug

## Software Systems Are Unpredictable and Unreliable

Gibbs, W. Wayt: "Software's Chronic Crisis", *Scientific American*, Sept. 1994, pp. 86–95.

> Studies have shown that for every six new large-scale software systems that are put into operation, two others are cancelled. The average software development project overshoots its schedule by half; larger projects generally do worse. And three quarters of all large systems are "operating failures" that either do not function as intended or are not used at all.
>
> Despite 50 years of progress, the software industry remains years–perhaps decades–short of the mature engineering discipline needed to meet the demands of an information-age society

omit

## Software Systems Are Unpredictable and Unreliable (cont.)

Wiener, Lauren Ruth *Digital Woes: Why We Should Not Depend Upon Software*

> Software products—even programs of modest size—are among the most complex artifacts that humans produce, and software development projects are among our most complex undertakings. They soak up however much time or money, however many people we throw at them.
>
> The results are only modestly reliable. Even after the most thorough and rigorous testing some bugs remain. We can never test all threads through the system with all possible inputs.

## Hardware Designs Are Unpredictable and Unreliable

Intel's President, Andy Grove, on comp.sys.intel Internet bulletin board:

> After almost 25 years in the microprocessor business, I have come to the conclusion that no microprocessor is ever perfect; they just come closer to perfection with each stepping. In the life of a typical microprocessor, we go thru [sic] half a dozen or more such steppings....

Michael Schrage, Washington Post Article (Dec 16, 1994)

> Pentium type problems will prove to be the rule—rather than the isolated, aberrant exceptions—as new generations of complex hardware and software hit the market. More insidious errors and harmful bugs are inevitable. That is the new reality.

## On the brink?

Commercial aviation has an excellent safety record BUT

- There are some signs that we are near the threshold where traditional approaches will no longer be able to deliver safe systems.
- The industry is rapidly adopting more and more complicated software systems, e.g. Integrated Modular Avionics (SC-182).

## A340 Incident (Near Heathrow on September 19, 1994)

- The commander's EFIS[1] map display symbology froze and lost all computed data [..].
- His MDCU[2] displayed the message 'PLEASE WAIT' and a *preflight* data load page. He was unable to obtain any other display.
- At roughly the same time, the copilot's EFIS and MCDU exhibited identical behavior.
- They 'dialed in' the ILS[3] using a back-up method,
- Then discovered they had some 2000 kgs of fuel less than expected (an erroneous indication).
- The autopilot's attempt to follow the glidepath resulted in unusually high pitch rates and so the autopilot was disconnected.
- The commander informed the tower they were having problems with the glideslope and requested an SRA (Surveillance Radar Approach)
- They landed without further incident; after taxiing in and shutting down, the fuel indications recovered.

[1] Electronic Flight Instrument System
[2] Multi-function Control and Display Unit
[3] Instrument Landing System

## AAIB[4] diagnosis

The reason for the wrong response of the autopilot and one flight director to the left turn demand was a *software error* ... This error was known to Airbus Industrie and corrective measures for this and several other software deficiencies ... have been issued ...

Post-flight analysis:

- The BITE data dump showed that the No 2 FMGEC had detected a CLASS 1 HARD failure within itself and a simultaneous fault within FMGEC No 1, but no hardware fault could be found.
- Airbus Industrie was aware of the double FMGS failure mode which had first emerged on the A320 series aircraft.
- According to Airbus Industrie, there are several ways in which the exchange of data and/or a problem in one computer can affect the other computer.
- Airbus Industrie has succeeded in radically reducing the frequency of double FMGS failures on the A320 series aircraft; they are also addressing the problem on the A330 and A340 series. [...]

[4] Air Accidents Investigation Branch of the UK Ministry of Transportation

## Radical Change In Digital Avionics for Commerical Aircraft

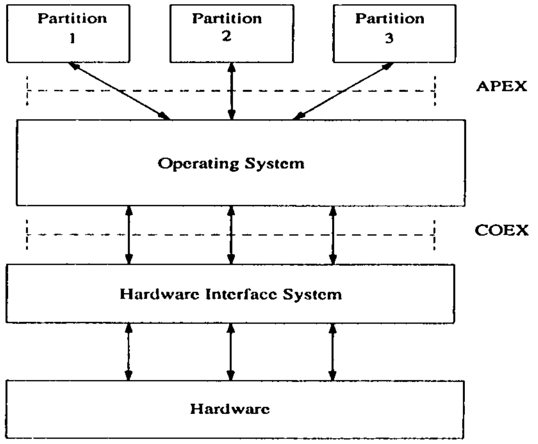
- Digital systems for commerical aircraft about to undergo a radical change
- The fundamental architecture of the avionics systems will move from "federated" to "integrated":
  - federated: Each aircraft function resides on its own computer. Aircraft functions are certified.
  - integrated: There is a shared computing resource on which multiple functions execute.

## The Motivation for this Change

According to ARINC 651, this approach [i.e. Integrated Modular Avionics (IMA)] will:

- Reduce life-cycle cost through
  - Reduced weight, power, and EE bay volume
  - Reduced unscheduled maintenance, reduced spares and increase parts commonality
- Reduce first cost and cost of service life support through
  - Reduced development, certification and production costs
  - Reduced avionics weight and increased payload volume
  - Flexibility to efficiently meet customer requirements and implement improvements

## IMA Picture



## IMA Issues

- IMA will introduce complex operating systems into avionics.
- Critical tasks and non-critical tasks will reside on same hardware.
- Task partitioning becomes *life-critical.*

*IMA needs formal methods technology*

## FAA Interest

Mike DeWalt, FAA National Resource Specialist For Aircraft Computer Software, encourgaged NASA Langley to pursue a research project aimed at applying Formal Methods to the partitioning problem:

> RTCA DO-178B recognizes partitioning as a crucial element to manage complexity.... However, the FAA has not developed any supportable criteria for establishing a partition. Although partitioning has been part of the certifications of the Gulfstream 4, Airbus A320, Boeing 747-400, Douglas MD-11 and is being used extensively in the certification of the Boeing 777 and the ongoing certification of the 412CF helicopter, *the assurance of these approvals has been non standardized and was largely based on individual engineering judgement. There is a critical need to establish supportable criteria for approving partitioning in avionics applications.*

## Why is Digital Design (SW and HW) so Hard to get Right?

- There's nothing to go wrong but the *design*
- Our intuition and experience is with continuous systems—but in digital system's the behavior is *discontinuous*
- Have to separately reason about or test millions of sequences of discrete state transitions
- Complexity exceeds our (unaided) intellectual grasp

## Classical vs. Computer Systems

| Classical Systems | Computer Systems |
|---|---|
| continuous state space | discrete state space |
| smooth transitions | abrupt transitions |
| finite testing<br>and interpolation OK | finite testing inadequate,<br>interpolation unsound |
| build to withstand additional stress | build to specific assumptions |
| mathematical modeling | prototyping and testing |
| predictable | surprising |

Why Formal Methods is Needed:

## Three Basic Approaches to Overcoming the Design Error Problem

- Testing (Lots of it)
- Design Diversity (i.e. Software Fault-Tolerance: N-version programming, recovery blocks, etc.)
- Fault Avoidance:
  - Formal Specification/Verification
  - Automatic Program Synthesis
  - Reusable Modules

## Life Testing

Basic Observation:

$10^{-9}$ probability of failure estimate for a 1 hour mission

Requires

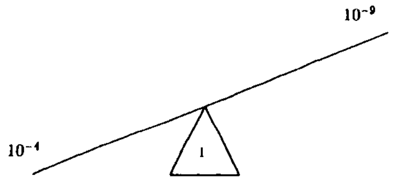
$> 10^9$ hours of testing

( $10^9$ hours = 114,000 years)

Design Diversity (i.e. Software Fault Tolerance)

- Use different versions of a program developed by different teams.
- Vote the outputs of the different versions.
- Hope that the redundant programs will fail independently or nearly so.

## What Enables Ultra-Reliability Quantification For Physical Failure



- The only thing that enables quantification of ultra-reliability for hardware systems with respect to physical failure is the:

  *INDEPENDENCE ASSUMPTION.*

## Can We Assume Independence for Multi-version Software?

- Unlike physical failures governed by the laws of physics, software errors are the products of human reasoning.
- Subjective arguments have been offered for and against the independence assumption.
- Subjective arguments for independence are not compelling enough to assume it as an axiom.

## Independence Model

### Experiments on Low Reliability Software (Knight and Leveson)

- 27 versions of a program subjected to 1,000,000 input cases.
- Observed average failure rate per input was 0.0007.
- Independence model was rejected at 99% confidence level.
- Other researchers have confirmed Knight-Leveson result.

## A More General Coincident Error Model?

Validating a general coincident error model would require more time than life testing a single system:

- Coincident errors must be observed
- Coincident errors be shown to correspond to the general model
- Extremely rare coincident errors must be observed to show model applies in ultra-reliable region

CONCLUSION:

For ultrareliable software, experiments cannot demonstrate that a general coincident error model is accurate.

## FM Versus Traditional Engineering of Digital Systems

| Method | Approach | Confidence |
|---|---|---|
| traditional | Build according to engineering judgement, test and measure failure rate | Empirical |
| formal methods | Design according to fundamental principles (*axioms*) and use *deduction* for refinement | Rational |

Although testing methods will be used in conjunction with formal methods the *primary confidence* in this system is *not* derived from the empirical data.

## What is Formal Methods

## Digital System Modeling and Prediction

- One way to predict behavior of a system is to construct a mathematical model and calculate it.
- For this to be effective, the model must be reasonably accurate and the calculations must be performed without error.
- For many continuous systems of traditional engineering, well-developed mathematical theories (e.g., Navier-Stokes equations for aerodynamics) are available.
- For computer systems, we must use discrete mathematics (logic, set theory) and build our own theories. Proofs of theorems take the place of numerical calculation.
- This is the essence of formal methods.

## Formal Methods

Formal Specification: Use of notations derived from formal logic to describe

- *assumptions* about the world in which a system will operate
- *requirements* that the system is to achieve
- a *design* to accomplish those requirements

Formal Verification: Use of methods from formal logic to

- *analyze* specifications for certain forms of consistency, completeness
- *prove* that the design will satisfy the requirements, given the assumptions
- *prove* that a more detailed design *implements* a more abstract one

## Formal Methods: Specification

- Instead of differential equations, the properties and behaviors are described using concepts from discrete mathematics: sets, graphs, partial orders, finite-state machines, and so on.
- The specification may be in traditional mathematical notation or in a computer-processable language:
  - enables typechecking
  - some automatic consistency checks
- Provides a clear, unambiguous specification.

## Formal Methods: Verification

- Formal Verification is more akin to proving theorems in geometry than to ordinary numerical calculation,
  - but like calculation it is performed according to strict rules
  - so one person can check the work of another
  - and computers can be used to automate some of the steps.
- The particular importance of formal methods to safety-critical systems is that they uniquely permit analysis of all the logical behaviors of a digital system [subject to some caveats].
- Total exploration is the only way to provide assurance that catastrophic failure does not lie hidden among the vast number of possible behaviors.

## How Is Total Exploration Possible?

Mathematical logic provides ways to reason about the properties of large or infinite collections of related things in a finite manner:

- *skolemization* allows us to draw conclusions for all values of a given variable by reasoning about a single representative symbolic constant
- *mathematical induction* allows us to deduce properties for all values of some ordered domain (e.g., the natural numbers, 0, 1, 2, ) by showing

(a) that the property is true for the least element (e.g., 0)
(b) that when it is true for all members up to some point (e.g., n), then it is also true for the next point
(e.g., n + 1)

## The Advantages of Formal Methods

- Enable faults in assumptions, requirements, design to be detected earlier than otherwise due to greater precision and explicitness early in the lifecycle.
- Enable faults to be detected with greater certainty than otherwise, by augmenting reviews with analysis.
- Can provide total coverage of selected, modeled properties.
- Guarantee absence of specified faults (subject to accuracy of modeling employed), because the calculations (proofs) can be checked mechanically (by a theorem prover)

## Conclusions

1. Formal methods offers the most intellectually defensible means of producing life-critical systems

2. Formal methods is the application of mathematical logic to digital system design and verification

# An Informal Introduction to *Formal Methods*

by
C. Michael Holloway
Paul S. Miner
Ricky W. Butler
Assessment Technology Branch

## Structure

- Definitions & Introduction to Basics of Logic
  - 30 minutes
  - Michael Holloway
- Application of Logic to Digital System Design
  - 30 minutes
  - Paul Miner
- *Break for 30 minutes*
- Detailed Example
  - 1 hour
  - Ricky Butler

## Purposes

- The five main purposes of these presentations are to
  - Provide a high-level overview of some of the foundations of formal methods
  - Demonstrate the variety of formal techniques that are available
  - Illustrate ways in which formal methods may be applied in various domains
  - Set the context for the remaining presentations at this workshop
  - Motivate future in-depth study of formal methods

## Non-Purposes

- These presentations are <u>not</u> intended to
  - Enable you to carry on a meaningful conversation with mathematical logicians
  - Present a balanced view of the relative capabilities of existing formal methods tools and techniques
  - Discuss all of the domains in which formal methods may be applied
  - Tell you all you need to know to become a formal methods expert
  - Put you to sleep

Omit

## What Does "Formal" Mean?

When many people think of the word "formal", they think of something like this:

## What Does "Formal" Mean?

(continued)

- Others think "formal" means one or more of the following things:
  - -- boring, uninteresting
  - -- difficult to understand, complicated
  - -- filled with many odd symbols and foreign letters

    $\forall \quad \exists \quad \supset \quad \lambda \quad \varnothing \quad \wedge \quad \vee \quad \aleph \quad \in \quad \equiv \quad \psi \quad \varphi$
- One of the goals of this workshop is to convince you that none of these perceptions is necessarily true

## What Does "Formal" Mean?

(continued)

- Webster's dictionary gives the following as one of the definitions of "formal":

  relating to, concerned with, or constituting the outward form of something as distinguished from its content
- A method is formal if its rules for manipulation are based on form (syntax) and not on content (semantics)
- Simple arithmetic is an example of such a method:

  -- 2 cats + 2 cats = 4 cats

  -- 2 mice + 2 mice = 4 mice

  -- 2x + 2x = 4x

  Validity of addition rests on its form alone — as long as like objects are added, it doesn't matter what those objects are.

  2 cats + 2 mice = ????

## What Does "Formal" Mean?

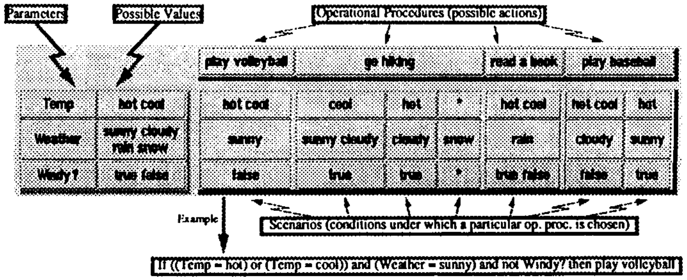
(continued)

| | | play volleyball | go hiking | | | read a book | play baseball | |
|---|---|---|---|---|---|---|---|---|
| Temp | hot cool | hot cool | cool | hot | * | hot cool | hot cool | hot |
| Weather | sunny cloudy rain snow | sunny | sunny cloudy | cloudy | snow | rain | cloudy | sunny |
| Windy? | true false | false | true | true | * | true false | false | true |

Is this table "formal" ?

# What Does “Formal” Mean?

(continued)



Parameters

Possible Values

Operational Procedures (possible actions)

| | | play volleyball | go hiking | | | read a book | play baseball | |
|---|---|---|---|---|---|---|---|---|
| Temp | hot cool | hot cool | cool | hot | * | hot cool | hot cool | hot |
| Weather | sunny cloudy rain snow | sunny | sunny cloudy | cloudy | snow | rain | cloudy | sunny |
| Windy? | true false | false | true | true | * | true false | false | true |

Scenarios (conditions under which a particular op. proc. is chosen)

Example

If ((Temp = hot) or (Temp = cool)) and (Weather = sunny) and not Windy? then play volleyball

Yes, it is.

# Symbolic Logic

- The majority of existing formal techniques are based on some flavor of symbolic logic

- Remainder of my talk will provide brief informal introduction to each of the following:

  -- Propositional Logic

  -- Predicate Logic

  -- Other Logics

- Along the way, we'll alsq discuss some of the difficulties in translating natural language into a logic

# Propositional Logic

- A *proposition* is a statement that is either true or false

  -- Today is Wednesday

  -- I have a son named David

  -- 2 + 2 = 5

  -- Fermat's last theorem is true

  -- This workshop is the most valuable workshop I've ever attended

- The possibility of assigning a truth value is all that is required; actual determination of the truth value is not required

- Questions, commands, etc., are not propositions

# Propositional Logic

(connectives)

- Propositions may be combined using the following *connectives*:

  -- **and** ($\wedge$): $P \wedge Q$ is true when both P and Q are true

  -- **or** ($\vee$): $P \vee Q$ is true when either P or Q is true

  -- **not** ($\neg$): $\neg P$ is true when P is false

  -- **If-then** ($\Rightarrow$, $\supset$): $P \supset Q$ is true except when P is true and Q is false

  -- **If-and-only-If** ($\equiv$): $P \equiv Q$ is true when P and Q have the same truth value

## Propositional Logic
(connectives vs. natural language)

- Please note that each connective has a precisely defined meaning
- This meaning does not always correspond to the natural language meaning
- The definition of **if-then** especially tends to confuse many people, because it does not require cause and effect

$$\left.\begin{array}{l} 2+2=4 \supset 2\times 2=4 \\ 2+2=5 \supset 2\times 2=4 \\ 2+2=5 \supset 2\times 2=5 \end{array}\right\}\text{are all true}$$

$$2+2=4 \supset 2\times 2=5 \quad \text{is false}$$

## Propositional Logic
(properties)

- Some properties (axioms & theorems) of propositional logic

-- **commutativity**: $P \wedge Q \equiv Q \wedge P$, $P \vee Q \equiv Q \vee P$

-- **associativity**: $(P \wedge Q) \wedge R \equiv P \wedge (Q \wedge R)$, same for $\vee$

-- **distributivity**: $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$
$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$

-- **De Morgan's laws**: $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$
$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$

-- **double negation**: $\neg\neg P \equiv P$

-- **implication definition**: $P \supset Q \equiv \neg P \vee Q$

## Propositional Logic
(inference rules)

- *Inference rules* permit reasoning (deducing conclusions based on the truth of certain premises)
- Some inference rules of propositional logic include

| *modus ponens* | *modus tollens* |
|---|---|
| $P \supset Q$ | $P \supset Q$ |
| $P$ | $\neg Q$ |
| $Q$ | $\neg P$ |

## Propositional Logic
(proofs)

- A **proof** of a given proposition establishes the truth of the proposition based only on axioms, theorems, and inference rules
- For propositional logic, truth tables provide a means of defining truth
- Propositional logic is

-- *complete*: everything that is true may be proven

-- *consistent (sound)*: nothing that is false may be proven

-- *decidable*: there exists a mechanical procedure for deciding whether any proposition is true or false

## Propositional Logic
(proof example)

- Show that $P \supset (Q \supset S) \equiv (P \wedge Q) \supset S$

- Proof

$$\begin{aligned}
P \supset (Q \supset S) &\equiv \neg P \vee (Q \supset S) && \textit{by implication definition}\\
&\equiv \neg P \vee (\neg Q \vee S) && \textit{by implication definition}\\
&\equiv (\neg P \vee \neg Q) \vee S && \textit{by associativity}\\
&\equiv \neg(P \wedge Q) \vee S && \textit{by De Morgan}\\
&\equiv (P \wedge Q) \supset S && \textit{by implication definition}\\
& && \textit{QED}
\end{aligned}$$

## Predicate Logic
(terms & propositions)

- Objects in predicate logic are denoted by expressions called *terms*:
  - -- Constants a, b, c, ... are terms
  - -- Variables u, v, w, ... are terms
  - -- If $t_1, t_2, ..., t_n$ are terms and *f* is a symbol that denotes a function of n arguments, then $f(t_1, t_2, ..., t_n)$ is a term
- In predicate logic, propositions are defined as follows:
  - -- *true* and *false* are propositions
  - -- If $t_1, t_2, ..., t_n$ are terms and *p* is a symbol that denotes a predicate of n arguments, then $p(t_1, t_2, ..., t_n)$ is a proposition

## Predicate Logic

- Propositional logic only permits reasoning about (true or false) complete sentences
- There is no way to reason about individual objects
  - -- There is a number n where $n \times n = n + n$
  - -- All cats have whiskers
- Predicate logic extends the formal language to permit reasoning about objects and the relationships between them
  - -- $\exists n: n \times n = n + n$
  - -- $\forall a: Cat(a) \supset Whiskers(a)$

## Predicate Logic
(sentences & connectives)

- Every proposition is also a *sentence* (or *formula*)
- Sentences may also be constructed using the connectives from propositional logic; if S and T are sentences, so are

  $S \wedge T,\ \ S \vee T,\ \ \neg S,\ \ S \supset T,\ \ S \equiv T$

  the meaning of each of these is as expected
- If x is a variable and S is a sentence, then the following are also sentences:
  - -- $\forall x: S$ true if S is true for all possible values of x
  - -- $\exists x: S$ true if S is true for at least one value of x

# Predicate Logic
(properties)

- Some properties of predicate logic

$$\neg\forall x\colon S \equiv \exists x\colon \neg S \qquad \neg\exists x\colon S \equiv \forall x\colon \neg S$$

$$\forall x\colon S \equiv \neg\exists x\colon \neg S \qquad \exists x\colon S \equiv \neg\forall x\colon \neg S$$

$$\forall x\colon \forall y\colon S \equiv \forall y\colon \forall x\colon S \qquad \exists x\colon \exists y\colon S \equiv \exists y\colon \exists x\colon S$$

- The following are true if x is not a free variable in S

$$S \vee \forall x\colon T \equiv \forall x\colon (S \vee T) \qquad S \vee \exists x\colon T \equiv \exists x\colon (S \vee T)$$

$$S \wedge \forall x\colon T \equiv \forall x\colon (S \wedge T) \qquad S \wedge \exists x\colon T \equiv \exists x\colon (S \wedge T)$$

# Predicate Logic
(translating from English)

- Translating natural language sentences into predicate logic is sometimes fairly simple

- Consider the following sentence:

  All humans are mortal

- Using the predicates human(x) and mortal(x), we have

  $\forall x$: human(x) $\supset$ mortal(x)

# Predicate Logic
(translating from English incorrectly)

- Translating natural language sentences into predicate logic is sometimes a little tricky

- Consider the following sentence:

  Apples and oranges are delicious and nutritious

- Using the predicates apple(x), orange(x), delicious(x), and nutritious(x), we might try the following:

  $\forall x$: (apple(x) $\wedge$ orange(x)) $\supset$ (delicious(x) $\wedge$ nutritious(x))

- Is this what we want?

# Predicate Logic
(translating from English correctly)

- This sentence does not have the intended meaning: it requires that an object be both an apple and an orange
- The following is what we really want:

  $\forall x$: (apple(x) $\vee$ orange(x)) $\supset$ (delicious(x) $\wedge$ nutritious(x))

- Here are a few more sentences for you to try at your leisure
  - -- Government workers are not always lazy
  - -- Testing never reveals the absence of errors
  - -- Only early registrants can attend the dinner at Captain George's
  - -- Nothing of importance was said at the workshop
  - -- A company creates good software if and only if it uses formal methods

# Predicate Logic

(translating from English: answers)

Government workers are not always lazy

$$\exists x\text{: government-worker}(x) \wedge \neg\text{lazy}(x)$$

Testing never reveals the absence of errors

$$\forall x\text{: testing}(x) \supset \neg\text{reveal-absence}(x)$$

Only early registrants can attend the dinner at Captain George's

$$\forall x\text{: dinner}(x) \supset \text{early-registrant}(x)$$

Nothing of importance was said at the workshop

$$\forall x\text{: important}(x) \supset \neg\text{said-at-workshop}(x)$$

A company creates good software if and only if it is uses FM

$$\forall x\text{: company}(x) \supset (\text{good-software}(x) \equiv \text{uses-formal-methods}(x))$$

# Predicate Logic

(proofs)

- A key concept in developing proofs in predicate logic is *skolemization*
  - -- Provides procedure for reducing a sentence into a propositional sentence that is true if the original sentence is true
  - -- Based on appropriate elimination of quantifiers
- Predicate logic is *complete* and *consistent,* but is only *semidecidable:*
  - -- There is a mechanical procedure that will terminate for all true sentences
  - -- This procedure may not terminate for all false sentences, but it will identify the sentence as false if it does terminate

# Predicate Logic

(proof example)

- Consider the sentence

$$\forall x\text{: } \exists y\text{: } y = x + 7$$

- We first replace the universally quantified variable x by the arbitrary constant a

$$\exists y\text{: } y = a + 7$$

- We may now remove the existential quantifier, giving the Skolem form for the sentence

$$y = a + 7$$

- This is easily satisfied by letting $y = a + 7$

$$a + 7 = a + 7$$

QED

# Other Logics

- *Many-sorted logic* extends predicate logic by dividing the universe from which variables and constants are chosen into several sorts (e.g., integers, reals, characters)
- *Higher order logic* (or *type theory*) allows functions to take functions as arguments and to return functions as values, and permits quantification over functions and predicates
- *Constructive logic* requires that a proof of existence of an object provides a procedure for constructing the object
- *Programming logics* provide rules for specifying and reasoning about imperative programs and program state
- *Modal logics, temporal logics, logics for partial functions*

## Higher Order Logic

(example)

- Consider the following statement:
  - -- There does not exist a list of all the possible functions that take a natural number as an argument and return a natural number as the result
- In a higher order logic, we might express this as follows:

$$
\begin{array}{l}
\neg\exists(\mathrm{list} : \text{function}[\text{nat} \rightarrow \text{function}[\text{nat} \rightarrow \text{nat}]]) \\
\quad \forall(g : \text{function}[\text{nat} \rightarrow \text{nat}]) \\
\quad\quad \exists(i : \text{nat}) \\
\quad\quad\quad \mathrm{list}(i) = g
\end{array}
$$

## Higher Order Logic

(example, continued)

- What does

$$\mathrm{list}(i) = g$$

mean?

- Two functions are defined to be equal if for every possible argument value, they return the same result
- This can be expressed in our notation as

$$
\begin{array}{l}
\forall(x : \text{nat}) \\
\quad \mathrm{list}(i)(x) = g(x)
\end{array}
$$

## Concluding Remarks

- Of the five main purposes mentioned at the beginning, I hope that my presentation has
  - -- Provided a high-level overview of some of the foundations of formal methods
  - -- Begun to set the context for the remaining presentations at this workshop
  - -- Perhaps motivated future in-depth study of formal methods
- The remaining presentations should continue to motivate and to set context, and should also
  - -- Demonstrate the variety of formal techniques that are available
  - -- Illustrate ways in which formal methods may be applied in various domains

# Application of Logic to Digital System Design

Paul S. Miner
May 10, 1995

Supporting text

The purpose of this talk is to illustrate what activities comprise the application of formal methods to digital system design. The examples given in this presentation are all drawn from real applications of formal methods.
A much better introduction to the issues involved is:
John Rushby: *Formal Methods and their Role in the Certification of Critical Systems*, Technical Report CSL-95-1, Computer Science Laboratory, SRI International, March 1995.
http://www.csl.sri.com/csl-95-1.html

# Outline

- Formal Specification
- Formal Proof Activities
- Degree of Rigor
- Obstacles to Getting Started

Supporting text

- The first part of the talk will address some of the issues involved in developing a formal specification. This will include illustrations of different styles of specification using examples drawn from real applications.
- The next section of the talk will address some of the different styles of refinement and proof in a formal development process.
- Formal methods can be applied with varying degrees of rigor. A taxonomy of levels of rigor will be presented.
- There are a number of obstacles for first time users to overcome. This section of the talk will enumerate some of the obstacles; hopefully, the remainder of this workshop will provide insights for overcoming them.

## Formal Specification

**Formal Specification:** Use of notations derived from formal logic to describe

- *assumptions* about the world in which a system will operate
- *requirements* that the system is to achieve
- the intended *behavior* of the system

**Styles of Specification** Different approaches are used for these descriptions:

- *Properties*—enumeration of assumptions and requirements
- *Functions*—express desired behavior or design descriptions
- *State-machines*—express desired behavior or design descriptions
- ...

Assumptions at one level become requirements at a lower level.

**Supporting text**

There are a number of different forms a formal specification can take. Often the style of specification will reflect the problem domain. Specifications describe some combination of *assumptions*, *requirements*, and *intended behavior* of the system. The description of the behavior can be high-level and abstract, or it may reflect a level in a design heirarchy.

## Example of Property-Based Specification (Fault-tolerant clock synchronization)

1. There is a $\rho \ll 1$ such that for any clock $C_p$ that is non-faulty during the interval from $t_1$ to $t_2$:

$$(1-\rho)(t_2 - t_1) \leq C_p(t_2) - C_p(t_1) \leq (1+\rho)(t_2 - t_1)$$

2. There is a $\delta$ such that if clocks $C_p$ and $C_q$ are non-faulty at time $t$, then:

$$|C_p(t) - C_q(t)| < \delta$$

**Supporting text**

The first property states that the function $C_p$ is a clock, that is, it provides a reasonable measure of the passage of time. The second states that any two non-faulty clocks in the collection are synchronized.

## Example of Functional Specification (Floating-point Operations)

IEEE floating-point arithmetic requires that each arithmetic operation *be performed as if it first produced a result correct to infinite precision and unbounded range, and then coerced this result to fit in the destination's precision.* [ANSI/IEEE STD 854-1987]

Supporting text

This is an example of an informal requirement. The easiest way to define arithmetic operations is to use a function based specification. With a sufficiently expressive specification language, this is a rather straightforward exercise.

## Floating-Point Operations (DECLARATIONS)

$$\begin{aligned} FP &= \text{the set of floating-point numbers} \\ Fin &= \text{the set of finitely valued floating-point numbers, } Fin \subset FP \\ M &= \text{the set of rounding modes} \\ \mathbf{R} &= \text{the set of real numbers} \\ fin1, fin2 &\in Fin \\ mode &\in M \end{aligned}$$

fp_to_real: $Fin \rightarrow \mathbf{R}$
real_to_fp: $R \times M \rightarrow FP$

Supporting text

The definition of floating-point operations assume that we have the following:

- A set of objects that represent the floating-point numbers
- A proper subset of the floating-point numbers that represents the finitely valued floating-point numbers
- A finite set of rounding modes
- Real Numbers
- Function `fp_to_real` is defined to convert any finitely valued floating-point number to a real number.
- Function `real_to_fp` takes two arguments: a real number and a rounding mode. It coerces the real into a floating-point representation in accordance with the rounding mode.

## Floating-Point Addition
## (Partial Specification)

For floating-point addition of two finitely valued arguments, $fin1$ and $fin2$, the expression

```
fp_to_real(fin1) + fp_to_real(fin2)
```

defines the infinitely precise result.

Supporting text

We can then define arithmetic operations *correct to infinite precision and unbounded range* using the corresponding operator in real arithmetic.

## Floating-Point Addition
## (CONTINUED)

The formal specification of floating-point addition for finitely valued arguments is:

```
fp_add_finite(fin1,fin2,mode) ≐
  real_to_fp( (fp_to_real(fin1) + fp_to_real(fin2)), mode)
```

Supporting text

This specification states precisely what a realization of floating-point addition for finite arguments must compute. It does not in any way determine how this function may be implemented. This definition also formally captures, in a natural way, the informally stated requirements of the standard.

## State-machine Specification

Design layers formalized as state machines

- State represents memory contents and hardware status
- Transition *function* defines state transitions

Interpretation maps lower level states into higher level states

$$\begin{tikzcd}
s_i \arrow[r, "N_i"] & t_i \\
s_{i+1} \arrow[r, "N_{i+1}"] \arrow[u, "Map"] & t_{i+1} \arrow[u, "Map"']
\end{tikzcd}$$

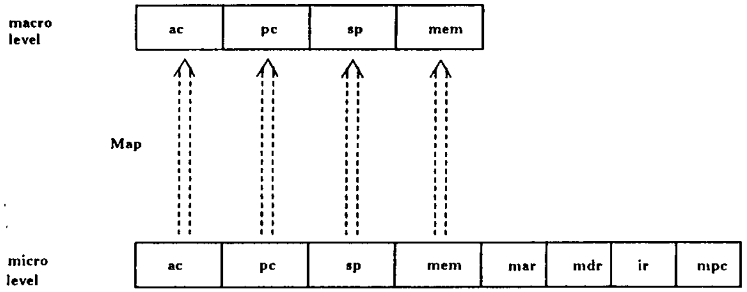
Need to show that diagram "commutes" to establish that layer $i + 1$ correctly implements layer $i$:

$$Map(N_{i+1}(s_{i+1})) = N_i(Map(s_{i+1}))$$

**Supporting text**

State-machine based specifications are useful for describing the behavior of digital systems. Systems are specified as a collection of states and a transition function. Different levels in a design hierarchy are related by abstraction mappings. Verification consists of showing that diagrams commute.

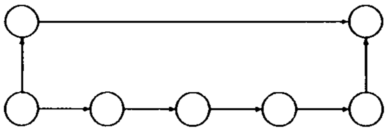
## Abstraction Mappings (Data Abstraction)



**Supporting text**

This slide illustrates a data abstraction similar to what would be used in a verification of a microprocessor. The macro-level defines the programmers view of the state; this include the accumulator, program counter, stack pointer and memory. The micro-level introduces some additional storage locations necessary for a correct realization of the macro-level specification; this includes the memory address register, memory data register, the instruction register, and microprogram counter.

## Temporal Abstraction



Supporting text

This diagram illustrate a simple temoral abstraction. The high-level state transition function is realized using a sequence of lower level state transitions.

## Formal Proof Activities

Use of methods from formal logic to

1. *analyze* specifications for certain forms of consistency, completeness
2. *prove* that specified behavior will satisfy the requirements, given the assumptions
3. *prove* that a more detailed design *implements* a more abstract one

Supporting text

A verification effort can consist of a number of different proof activities. There are a number of formal techniques for analyzing specifications. These help provide assurance that the specification describes what we intend. Proof techniques are useful for showing that specified behavior exhibits desired prpoerties. Similarly, proof can be used to relate different levels in a design hierarchy.

## Formal Analysis of a Specification (1)

One property we might want to prove about the functions real_to_fp and fp_to_real from the floating-point addition specification is that for every finitely valued floating-point number and every rounding mode:

$$\texttt{real\_to\_fp}(\texttt{fp\_to\_real}(fin), mode) = fin$$

Supporting text

In order to ensure that the specification captures what we intend, we postulate certain properties that should be true of the specification. Formal techniques allow us to prove a spcification posseses these properties. This is sometimes referred to as formal validation.
The example presented here states that the function converting real numbers to floating-point representations should invert fp_to_real, irrespective of the rounding mode. As stated, the property is not true. Additional restrictions are required, because the IEEE-854 floating-point stadard allows redundant encodings.

## Verification of Fault-Tolerant Algorithms (2)

**Top-level:** Properties that algorithm should possess

**Lower-level:** Abstract description of the algorithm and underlying assumptions

**Prove:** The algorithm satisfies desired properties given the assumptions

Supporting text

As an example, the top-level requirement for fault-tolerant clock synchronization is an enumeration of properties. The verification of an algorithm consists of showing that the algorithm, given certain assumptions about the environment, ensures all the properties.

## Design Verification (3)

**Top-level:** Abstract description of system (and assumptions)

**Lower-level:** Detailed description of system (and assumptions)

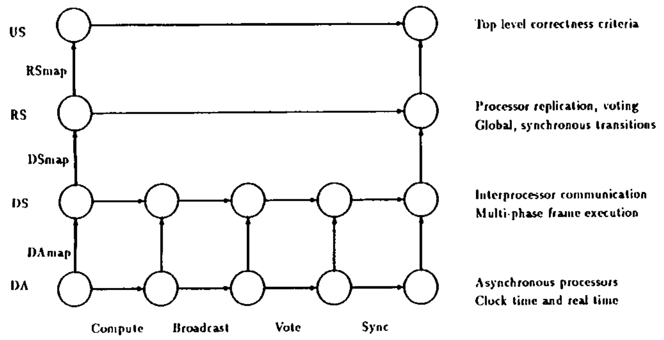
**Prove:** The detailed system description has the same behavior as the abstract description given the assumptions and an abstraction function relating the two systems.

Supporting text

The next example will give a high-level overview of design verification.

## Design Refinement and Proof (Example: Reliable Computing Platform)

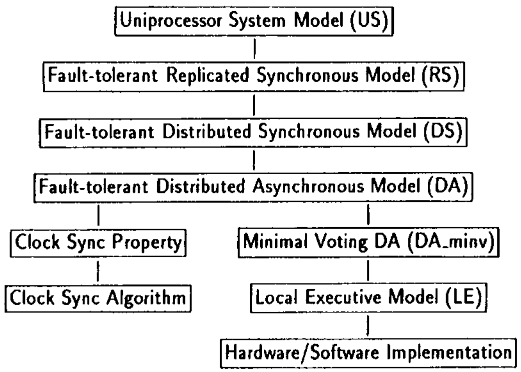
Single-frame state transition divided into four phases


US
RSmap
RS
DSmap
DS
DAmap
DA
Compute
Broadcast
Vote
Sync
Top level correctness criteria
Processor replication, voting
Global, synchronous transitions
Interprocessor communication
Multi-phase frame execution
Asynchronous processors
Clock time and real time

Supporting text

This diagram illustrates four levels in the design hierarcy of the Reliable Computing Platform. The top-level specification provides the applications programmer a high-level view of a single, ultra-reliable computer. Lower levels in the diagram introduce the necessary replication and redundancy management to mask physical failures. These levels all use the state-machine approach.

## Hierarchical Specification of the Reliable Computing Platform


Uniprocessor System Model (US)
Fault-tolerant Replicated Synchronous Model (RS)
Fault-tolerant Distributed Synchronous Model (DS)
Fault-tolerant Distributed Asynchronous Model (DA)
Clock Sync Property
Minimal Voting DA (DA_minv)
Clock Sync Algorithm
Local Executive Model (LE)
Hardware/Software Implementation

**Supporting text**

This slide shows all the levels in the RCP design verification effort. The top four levels are the same as the previous slide. This slide also illustrates how the clock synchronization is addressed. The top four levels assume the clock synchronization properties. At a lower level, a proof is given showing that an algorithm guarantees the properties. This hierarchy illustrates how different techniques may be employed in one large verification effort.

## Correctness of Specification

How do we know that a specification is what we intend?

- At intermediate levels in the design hierarchy, the specification is a refinement of higher-level requirements
  - If the higher levels are related by proof, then the requirements at this level are sufficient to ensure the high level requirements
- At the top-level, we must resort to review.
  - The presence of a formal specification enables formal challenges. A review can identify additional properties that a design must possess. Formal verification techniques can then be used to either prove the design already possesses the property or reveal its absence.
- At the bottom levels in a design hierarchy, we must show that the assumptions are consistent.

**Supporting text**

Formal techniques can ensure that intermediate steps in the design process are carried out correctly, but we are still faced with the problem of correctly stating the top-level specification. Formal specification techniques enable us to analyze the top level specification in a more rigorous fashion, thus providing confidence that the specification is what we intend. However, it is still necessary to review the top-level to ensure that it matches expectations.

## Levels of Rigor

**Level 1:** Specification (and proof) using conventional mathematical notation.

**Level 2:** Specification using a formal specification language with manual proofs.

**Level 3:** Specification in a formal language with automatically checked or generated proofs.

Supporting text

- Level 1 formal techniques allow freedom of expressiveness, but very little potential for mechanized support. The logical system need not be precisely defined and proofs need not explicitly address all the details.
- In Level 2, the specification language has a well defined syntactic structure and proofs proceed according to well defined rules. There is potential for some mechanized support to ensure that specifications adhere to syntactic rules and that some proof rules are applied correctly.
- In Level 3. the language and proof system have tightly integrated mechanized support. Sometimes, expressiveness of logic is restricted to allow more efficient/automatic mechanized support. Makes it feasible to formally analyze systems with greater precision than is possible with paper-and-pencil approaches.

## Mechanized Support for Formal Methods

**General Purpose Theorem Provers:** Specification and Proof using the logic supported by theorem prover. Proofs are semi-automatic. Many of the steps are automated, but some require insight.

Examples include: PVS, HOL, Nuprl, Nqthm/Acl2, IMPS, . . .

**Specialized Approaches:**

**Model Checking:** Fully automatic proofs that state machine description of hardware possess specified properties. Specifications given in a decidable temporal logic. An example system is SMV. Tools employing related techniques include COSPAN and Mur$\phi$.

**Design Derivation:** Design proceeds from a behavioral level description to a hardware design via a series of *correctness preserving refinements*. Example systems include DDD and DRS.

**Software:** Machine checked verification of Ada code (Penelope). Decision tables for specification of software requirements (TableWise).

Supporting text

There are a number of chioces for level 3 application of formal methods. Specialized tools are useful for addressing some areas of the design space, but general purpose systems allow for addressing a number of different verification techniques within a single framework. Current research efforts include developing approaches to combine general purpose theorem provers with the more specialized techniques.

## Obstacles to getting started

- Diversity of tools
  - Choice of proper tool is difficult
  - Over 50 notations, methods, and tool listed on WWW Formal Methods Virtual Library
    http://www.comlab.ox.ac.uk/archive/formal-methods.html
  - Many domain specific tools and techniques
  - Little support for combining results from different tools
  - Some efforts currently in progress
- Immaturity of Tools
  - Most tools are research prototypes
- Lack of sufficient Libraries
- Education

Supporting text

The diversity of tools and techniques is both a blessing and a curse. There are a number of domain specific approaches that are appealing, but a significant amount of expertise is required to make an intelligent choice.

OMIT

# Flight Schedule Database Example

by

Ricky W. Butler
NASA, Langley Research Center

May 19, 1995

# Flight Schedule Example

## Requirements for an Airport Flight Schedule Database

- The flight schedule database shall store the scheduling information associated with all departing and arriving flights. In particular the database shall contain:
  - departure time and gate number
  - arrival time and gate number
  - route (i.e. navigation way points)

  for each arriving and departing flight.
- There shall be a way to retrieve the scheduling information given a flight number.
- It shall be possible to add and delete flights from the database.

## Formal Requirements Specification

- How do we represent the flight schedule database mathematically?

1. a set of ordered pairs of flight numbers and schedules. Adding and deleting entries via set addition and deletion
2. function whose domain is all possible flight numbers and range is all possible schedules. Adding and deleting entries via modification of function values.
3. function whose domain is only flight numbers currently in database and range is the schedules. Adding and deleting entries via modification of the function domain and values.

Note: The choice between these is strongly influenced by the verification system used.

## Getting Started

Let's start with approach 2:

function whose domain is all possible flight numbers and range is all possible schedules. Adding and deleting entries via modification of function values.

In traditional mathematical notation, we would write:

$$\text{Let } N = \text{set of flight numbers}$$
$$S = \text{set of schedules}$$
$$D : N \longrightarrow S$$

where $D$ represents the database and $S$ represents all of the schedule information.

Note that the details have been *abstracted away.* This is one of the most important steps in producing a good formal specification.

## Specifying the Flight Schedule Database

$$D : N \longrightarrow S$$

**How do we indicate that we do not have a flight schedule for all possible flight numbers?**

We declare a constant of type $S$, say "$u_o$", that indicates that there is no flight scheduled for this flight number.

**Now can define an empty database. In traditional notation, we would write:**

$$empty\_database : N \longrightarrow S$$

$$empty\_database(flt) \equiv u_o$$

$$\forall\ flt \in N$$

## Accessing an Entry

**Let** $N$ **= set of flight numbers**

$S$ **= set of schedules**

$D$ **= set of functions** $: N \longrightarrow S$

$\forall d \in D$ **and** $flt \in N$.

$$find\_schedule : D \times N \longrightarrow S$$

$$find\_schedule(d, flt) = d(flt)$$

**Note that** *find_schedule* **is a higher-order function since its first argument is a function.**

## Specifying Adding/Deleting an Entry

**Let** $N$ **= set of flight numbers**

$S$ **= set of schedules**

$D : N \longrightarrow S$

$u_o \in S$

$D$ **= set of functions** $: N \longrightarrow S$

$\forall d \in D,\ \forall flt \in N,\ \forall sched \in S$

$$add\_flight : D \times N \times S \longrightarrow D$$

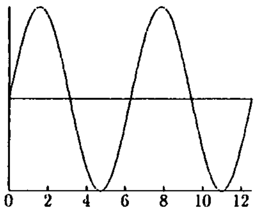
$$add\_flight(d, flt, sched)(x) = \begin{cases} d(x) & \text{if } x \neq flt \\ sched & \text{if } x = flt \end{cases}$$

$$delete\_flight : D \times N \longrightarrow D$$

$$delete\_flight(d, flt)(x) = \begin{cases} d(x) & \text{if } x \neq flt \\ u_o & \text{if } x = flt \end{cases}$$

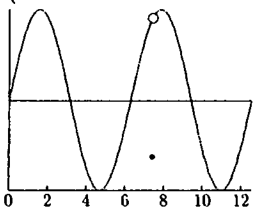
## The WITH Notation

**sin(x):**



$$\textbf{sin(x) WITH [7.4 := -0.60 ]} = \begin{cases} \textbf{-0.60} & \textbf{if } x = 7.4 \\ sin(x) & \textbf{otherwise} \end{cases}$$

**sin(x) WITH [7.4 := -0.60 ]**



## Complete Spec (Omitting Function Signatures)

Let $N$ = set of flight numbers
$S$ = set of schedules
$D$ = set of functions : $N \longrightarrow S$
$\forall d \in D,\ \forall flt \in N,\ \forall sched \in S$

$find\_schedule(d, flt) = d(flt)$

$add\_flight(d, flt, sched)(x) = d$ **WITH** $[flt := sched]$

$delete\_flight(d, flt)(x) = d$ **WITH** $[flt := u_o]$

**Can test spec with some putative theorems:**

LEMMA (*putative 1*) : $find\_schedule(add\_flight(d, flt, sched), flt) = sched$
LEMMA (*putative 2*) : $delete\_flight(add\_flight(d, flt, sched), flt) = d$

## Attempted Verification Of Putative 2 Reveals a Problem

LEMMA (*putative 2*): $delete\_flight(add\_flight(d, flt, sched), flt) = d$
**Proof:**

$delete\_flight(add\_flight(d, flt, sched), flt) =$

$delete\_flight(d$ **WITH** $[flt := sched]) =$

$d$ **WITH** $[flt := sched]$ **WITH** $[flt := u_o] =$

$d$ **WITH** $[flt := u_o] = ??$

**But there is no way to reach $d$, because**

$d$ **WITH** $[flt := u_o] \neq d$

**unless $d(flt) = u_o$.**

**This is only true if the $flt$ is currently not scheduled in the flight database.**

## Verification Reveals Oversight

- **We realize that we only want to add a flight with flight number $flt$, if one is not already in the database.**
- **If $flt$ is already in the database, we probably need the capability to change it.**

**Thus, we modify $add\_flight$ and create a new function $change\_flight$:**

## Verification Reveals Oversight (Cont.)

Let $N$ = set of flight numbers
$S$ = set of schedules
$D$ = set of functions : $N \longrightarrow S$
$\forall d \in D,\ \forall flt \in N,\ \forall sched \in S$

$scheduled?(d, flt) : boolean = d(flt) \neq u_o$

```
add_flight(d, flt, sched) =
        IF scheduled?(d, flt) THEN d
        ELSE d WITH [flt := sched] ENDIF

change_flight(d, flt, sched) =
        IF scheduled?(d, flt) THEN d WITH [flt := sched]
        ELSE d ENDIF
```

## Putative 2 Proof After Correction

**LEMMA** (*putative 2*): **NOT** $scheduled?(d, flt) \supset$
$delete\_flight(add\_flight(d, flt, sched), flt) = d$

**Proof:**

$delete\_flight(add\_flight(d, flt, sched), flt)$

$= delete\_flight($ **IF** $scheduled?(d, flt)$ **THEN** $d$
**ELSE** $d$ **WITH** $[flt := sched]$ **ENDIF** )

$= delete\_flight(d$ **WITH** $[flt := sched])$

$= d$ **WITH** $[flt := sched]$ **WITH** $[flt := u_o]$

$= d$ **WITH** $[flt := u_o]$

$= d$ (because **NOT** $scheduled?(d, flt) \supset d(flt) = u_o$ )

## A Minor Problem

**To check our new function schedule? we postulate the following putative theorem:**

**SchedAdd: LEMMA** $scheduled?(add\_flight(d, flt, sched), flt)$

**Proof:**

$scheduled?(add\_flight(d, flt, sched)) =$

$scheduled?($ **IF** $scheduled?(d, flt)$ **THEN** $d$
**ELSE** $d$ **WITH** $[flt := sched]$ **ENDIF** $=$

**IF** $d(flt) \neq u_o$ **THEN** $d(flt) \neq u_o$
**ELSE** $d$ **WITH** $[flt := sched](flt) \neq u_o$ **ENDIF** $=$

$d$ **WITH** $[flt := sched](flt) \neq u_o$

$sched \neq u_o$

**which is not provable because nothing prevents** $sched = u_o$.

## A Minor Problem Repaired

**We then realize that our specification does not rule out the possibility of assigning a "$u_o$" schedule to a real flight**

**Let** $N =$ **set of flight numbers**
$S =$ **set of schedules**
$S^* =$ **set of schedules not including** $u_o$
$D =$ **set of functions** $: N \longrightarrow S$
$\forall d \in D,\ \forall flt \in N,\ \forall sched \in S^*$
$find\_schedule : D \times N \longrightarrow S$
$add\_flight : D \times N \times S^* \longrightarrow D$
$change\_flight : D \times N \times S^* \longrightarrow D$
$delete\_flight : D \times N \longrightarrow D$

**This type of trivial problem is usually not manifested until when one attempts a mechanical (i.e. level 3) verification.**

## Another Example of a Putative Theorem

$$
\begin{array}{ll}
 & (\forall i : flt_i \neq flt) \wedge \\
 & \\
 & find\_schedule(d_0, flt) = sched \wedge \\
 & d_1 = add\_flight(d_0, flt_1, sched_1) \wedge \\
 & d_2 = add\_flight(d_1, flt_2, sched_2) \wedge \\
 & \vdots \\
 & d_n = add\_flight(d_{n-1}, flt_n, sched_n) \\
\supset & \\
 & find\_schedule(d_n, flt) = sched
\end{array}
$$

- **Formal methods can establish that even in the presence of an *arbitrary* number of operations a property holds.**
- **Testing can never establish this.**

## Some Observations

- Our specification is abstract. The functions are defined over infinite domains.
- As one translates the requirements into mathematics, many things that are usually left out of English specifications are explicitly enumerated.
- The formal process exposes ambiguities and deficiencies in the requirements.
- Putative theorem proving and scrutiny reveals deficiencies in the formal specification.

## PVS Spec

```
flight_sched3: THEORY
BEGIN

  N : TYPE                      % flight numbers
  S : TYPE                      % schedules
  D : TYPE = [N -> S]           % flight database

  u0: S                         % unscheduled

  S_good   : TYPE = {sched: S | sched /= u0}

  flt   : VAR N
  d     : VAR D
  sched : VAR S_good

  emptydb(flt): S = u0

  find_schedule(d, flt): S = d(flt)

  scheduled?(d,flt): boolean = d(flt) /= u0
```

```
  add_flight(d, flt, sched): D =
       IF scheduled?(d,flt) THEN d
       ELSE d WITH [flt := sched] ENDIF

  change_flight(d, flt, sched): D =
       IF scheduled?(d,flt) THEN d WITH [flt := sched]
       ELSE d ENDIF

  delete_flight(d, flt): D = d WITH [flt := u0]


  putative2  : LEMMA NOT scheduled?(d,flt) IMPLIES
                        delete_flight(add_flight(d,flt,sched),flt) = d

  SchedAdd   : LEMMA scheduled?(add_flight(d,flt,sched),flt)


END flight_sched3
```

## Introduction to a PVS Proof

- Illustrative proof
- The single command GRIND proves it automatically

```
putative2 :

  |-------
{1}   (FORALL (d: D, flt: N, sched: S_good):
         NOT scheduled?(d, flt)
           IMPLIES delete_flight(add_flight(d, flt, sched), flt) = d)

Rule? (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
putative2 :

  |-------
{1}   scheduled?(d!1, flt!1)
{2}   delete_flight(add_flight(d!1, flt!1, sched!1), flt!1) = d!1

Rule? (EXPAND "add_flight" )
Expanding the definition of add_flight,
this simplifies to:
putative2 :

  |-------
[1]   scheduled?(d!1, flt!1)
{2}   delete_flight(IF scheduled?(d!1, flt!1) THEN d!1
                ELSE d!1 WITH [flt!1 := sched!1]
                ENDIF,
                flt!1)
        = d!1

Rule? (LIFT-IF )
Lifting IF-conditions to the top level,
this simplifies to:
putative2 :

  |-------
[1]   scheduled?(d!1, flt!1)
{2}   IF scheduled?(d!1, flt!1) THEN delete_flight(d!1, flt!1) = d!1
      ELSE delete_flight(d!1 WITH [flt!1 := sched!1], flt!1) = d!1
      ENDIF

Rule? (ASSERT)
Simplifying, rewriting, and recording with decision procedures,
```

```
this simplifies to:
putative2 :

  |-------
[1]   scheduled?(d!1, flt!1)
{2}   delete_flight(d!1 WITH [flt!1 := sched!1], flt!1) = d!1

Rule? (EXPAND "delete_flight" )
Expanding the definition of delete_flight,
this simplifies to:
putative2 :

  |-------
[1]   scheduled?(d!1, flt!1)
{2}   d!1 WITH [flt!1 := sched!1] WITH [flt!1 := u0] = d!1

Rule? (EXPAND "scheduled?" )
Expanding the definition of scheduled?,
this simplifies to:
putative2 :

  |-------
{1}   d!1(flt!1) /= u0
[2]   d!1 WITH [flt!1 := sched!1] WITH [flt!1 := u0] = d!1
```

```
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
Q.E.D.


Run time  = 1.16 secs.
Real time = 61.49 secs.

Wrote proof file /airlab/home/rwb/fm/wkshp/pvs/flight_sched3.prf
NIL
>
```

## New Requirement

**"Two flights are not to be scheduled at the same gate at the same time!"**

**Introduce refinement of schedule:**

```
N  : TYPE
Date_and_time: TYPE
Gate_nums: TYPE
A_or_D: TYPE = (arriving,departing)
Way: TYPE

S: TYPE = [# % RECORD
          departure_tm: Date_and_time,
          arrival_tm: Date_and_time,
          dep_gate: Gate_nums,
          arr_gate: Gate_nums,
          arr_or_dep: A_or_D,
          nav_route: Way
          #] % END RECORD
```

## Simplified Problem

Often it is useful to solve a simplified problem before you tackle the big problem. So let's only work with departing flights:

```
N  : TYPE
Date_and_time: TYPE
Gate_nums: TYPE

S: TYPE = [# % RECORD
          departure_tm: Date_and_time,
          dep_gate: Gate_nums,
          #] % END RECORD
```

The requirement states that "two flights are not to be scheduled at the same gate at the same time":

```
same_time(sched1, sched2): boolean
```

## New Requirement Continued

We also need to introduce concept of "scheduled at the same gate at the same time":

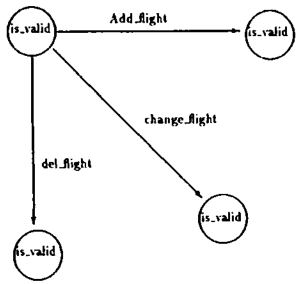
```
overlapped(sched1,sched2): boolean = dep_gate(sched1) = dep_gate(sched2)
                                     AND same_time(sched1,sched2)
```

We would like to establish that the operations on the database will never result in an overlapped situation.

In other words, we want to establish an *invariant:*

```
is_valid(d: D): boolean = (FORALL (flt1,flt2: N): flt1 /= flt2 AND
           scheduled?(d,flt1) AND scheduled?(d,flt2) IMPLIES
                NOT overlapped(find_schedule(d,flt1), find_schedule(d,flt2)))
```

## Database System as a State Machine



Need to establish that all of the "operations" maintain the invariant. For example,

```
add_flight_is_valid: LEMMA
     (FORALL (d: D, flt: N, sched: S_good):
           is_valid(d) IMPLIES is_valid(add_flight(d,flt,sched)));
```

Of course, add_flight must be modified to insure that this is true:

## Add_flight Modified To Maintain Invariant

```
gate_in_use_at_time(d,sched): boolean =
       (EXISTS flt: scheduled?(d,flt) AND overlapped(sched,d(flt)))

add_flight(d, flt, sched): D =
     IF scheduled?(d,flt) OR gate_in_use_at_time(d,sched) THEN d
     ELSE d WITH [flt := sched]
     ENDIF
```

Thus, we have modeled the database as a finite state machine and the functions add_flight, change_flight, and delete_flight are operations on the state machine.

## State Machines and PVS Type System

By creating a predicate subtype of the type D:

```
Valid_db:    TYPE = {d: D | is_valid(d)}
```

and modifying the signatures of add_flight, change_flight, and delete_flight, e.g.

```
add_flight(vd: Valid_db, flt, sched): Valid_db =
    IF scheduled?(vd,flt) OR gate_in_use_at_time(vd,sched) THEN vd
    ELSE vd WITH [flt := sched]
    ENDIF
```

PVS will automatically generate the "invariant" lemmas that must proved (called TCC's).

- is just a particular case of the more general TCC mechanism
- illustrates how a mechanized specification language can provide much stronger typechecking than traditional programming languages

## Conclusions

- With formal methods a clear, unambiguous, abstract specification can be constructed.
- Mechanized formal methods allows you can CALCULATE (prove) whether the specification has certain properties.
- These calculations can be done early in the lifecycle on abstract descriptions.
- And they can cover ALL the case,.

# Overview of NASA Langley's Formal Methods Program

by

Ricky W. Butler

NASA Langley Research Center

May 10, 1995

## Goals Of Our Formal Methods Program

- To make formal methods practical for use on life critical systems developed in the United States.
- To orchestrate the transfer of this technology to industry through use of carefully designed demonstration projects.

## NASA Langley's Research Strategy

- Build strong team at LaRC.
- Leverage huge investment of ARPA and National Security Agency through use of FM technology that they nurtured.
- Use contracts with the leading U.S. formal methods research teams.
- Establish joint projects with FM researchers and established aerospace engineering teams (e.g. Rockwell Collins, Honeywell, Boeing).
- Concentrate on the technically challenging areas of digital design (especially in avionics) that are currently beyond the state-of-the-practice.
- Initiate demonstration projects in problem domains where current formal methods are adequate.

## Although Tool Development Has Not Been Emphasized ...

- NASA Langley has not sponsored the development of any general-purpose theorem provers.
- However, the technology transfer projects have lead to significant improvements in the Prototype Verification System (PVS) theorem prover that SRI International is developing.
- Some domain-specific tools are being sponsored:
  - VHDL-analysis tool [ORA]
  - Tablewise [ORA]
  - Derivational Reasoning System (DRS) [Derivation Systems Inc.]

OMIT

## Scope of Program is Large

- Formal Methods appropriate for one problem domain may be totally inappropriate for other problem.
- Specific domains in which our program has concentrated:
  - architectural-level fault tolerance,
  - clock-synchronization,
  - interactive consistency,
  - design of hardware devices such as microprocessors, memory management units, DMA controllers,
  - asynchronous communication protocols,
  - design and verification of application-specific integrated circuits (ASICS),
  - Space Shuttle software,
  - navigation software,
  - decision tables,
  - railroad signaling systems.
- We are also interested in applying formal methods to many different portions of the life-cycle, such as (1) requirements analysis, (2) high-level design, (3) detailed design, and (4) implementation.

## Technology Transfer Strategy

- Get involved with (i.e. influence) standards activities (e.g. NIST, DO178B).
- Work with/influence FAA.
- Demonstrate techniques on realistic designs.
- Sponsor joint research projects using both formal methods contractors and aerospace contractors.

## Technology Transfer Projects

1. Rockwell Collins AAMP5
2. Boeing PIU Project
3. Space Shuttle Jet-Select Project
4. Honeywell Navigation
5. Charles Stark Draper FTPP Scoreboard Project
6. Honeywell Software Development (Tinkerbell)
7. Union Switch and Signal
8. Rockwell Collins AAMP-FV
9. Honeywell SafeBus Project

## Transfer of Formal Methods to Industry Has Been Slow in the Past

Why?

- Long learning curve:
  - FM not part of traditional engineering curriculum
  - the tools are not production-quality
  - the tools have few or no examples for specific problem domains.
- Many competing methods: the best are now only beginning to stand out.
- Cost/benefit relationship not favorable for many applications: situation has improved considerably over last 5-10 years, and improving this relationship is the major goal of our research program.
- Large investment by DARPA and NSA in tool development but did not fund technology transfer projects.

## Also Libraries Are Needed

There is a sizable effort associated with the development of the background mathematical theories needed for a particular problem domain.

- Libraries would be reusable and in the long run "cost-effective"
- But, initial costs are a deterrent for industry.

Therefore, one of the goals of the NASA Langley program is to build a large body of background theories needed for aerospace applications. Langley has helped develop PVS libraries:

- bitvectors library for hardware verification
- IEEE floating point standard
- finite sets
- majority
- summations

## Finelli/Di Vito Inject Formal Methods Into DO178B

George Finelli, a member of the verification subcommittee of the RTCA group that developed the DO-178B standard and Ben Di Vito wrote a section on formal methods that was included in DO-178B:

> 12.3.1. Formal Methods: Formal methods involve the use of formal logic, discrete mathematics, and computer-readable languages to improve the specification and verification of software. These methods could produce an implementation whose operational behavior is known with confidence tobe within a defined domain. In their most thorough application, formal methods could be equivalent to exhaustive analysis of a system with respect to its requirements. Such analysis could provide:
>
> - Evidence that the system is complete and correct with respect to its requirements.
> - Determination of which code, software requirements or ... satisfy the next higher level of software requirements.
>
> ... The goal of applying formal methods is to prevent and eliminate requirements, design and code errors throughout the software development processes. Thus, formal methods are complementary to testing....
>
> Formal methods may be applied to software development processes with consideration of these factors:
>
> **Levels of the design refinement** : The use of formal methods begins by specifying software high-level requirements in a formal specification language and verifying by formal proofs that they satisfy system requirements, especially constraints on acceptable operation. The next, lower level of requirements are then shown to satisfy the high-level requirements. Performing this process down to the Source Code provides evidence that the software satisfies system requirements. Application of formal methods can start and stop with any consecutive levels ...
>
> **Coverage of software requirements and software architecture** : Formal methods may be applied to software requirements that: (1) Are safety-related, (2) Can be defined by discrete mathematics, (3) involve complex behavior, such as concurrency, distributed processing, redundancy management, and synchronization.
>
> **Degree of rigor** : Formal methods include these increasingly rigorous levels: ...
>
> The use of formal specifications alone forces requirements to be unambiguous. Manual proof is a well-understood process that can be used when there is little detail. Automatically checked or generated proofs can aid the human proof process and offer a higher degree of dependability, especially for more complicated proofs.

## RTCA committees

Currently, members of the Langley staff are involved in:

- SC-180 (Airborne Electronic Hardware)
- SC-182 (Minimal Operating Performance Standard for an Airborne Computer Resource)

## Some Lessons Learned

- Modern formal specification languages such as PVS (which support higher order logic and a rich type system) provide a means of writing specifications that *can be read and understood* by engineers.
- Formal specification alone can reveal bugs not found by traditional V&V techniques. Formal Verification can uncover additional errors and provide increased confidence in the correctness of the system design.
- There is a sizable up-front cost associated with transferring formal methods into a commercial company. The two major components of this cost are:
  1. Training the industrial experts to use the formal techniques (especially in verification).
  2. The formal methods experts must learn the problem domain in detail (to develop effective formal methods).

## Some Lessons Learned (Continued)

- Finding a person inside the company who is knowledgeable of the problem domain, is a proponent of formal methods, and is an active participant in the project is essential.
- The formal methods researchers must be willing to adapt their methods to the problem domain rather than fight to change the existing methodologies to conform to their needs.
- Without the significant increases in hardware speed, level 3 formal methods would still be impractical.
- Efficient automatic deduction can greatly facilitate the useability of a theorem prover.

## NASA Langley's Formal Methods (FM) Program

- The tremendous scientific *potential* of formal methods has been recognized by theoreticians for a long time, BUT
- The formal techniques had remained the province of a few academians with only a few exceptions such as the Transputer and CICS.
- Starting in 1991, NASA Langley initiated several aggressive projects designed to move FM into productive use in the aerospace community:
  - Boeing PIU Project (1991)
  - Charles Stark Draper FTPP Scoreboard Project (1991)
  - Allied Signal Hybrid Fault Models (1992)
  - Shuttle Tile Project (1992)
  - Space Shuttle Jet-Select Project (1993)
  - Honeywell Navigation (1993)
  - Rockwell Collins AAMP5 (1993)
  - Honeywell Software Development: Tinkerbell (1994)
  - Union Switch and Signal (1994)
  - Rockwell Collins AAMP-FV (1995)
  - Honeywell SafeBus Project (1995)
- NASA's program has advanced FM in the United States to the point where commercial exploitation of FM is imminent.

# Session 2: LaRC-sponsored Industrial Applications

**Ricky W. Butler, Chair**

- **The AAMP5/AAMP-FV Project** by *Steve Miller*, Rockwell-Collins
- **Union Switch & Signal Project**, by *Joe Profeta*, Union Switch & Signal; and *Doug Hoover*, Odyssey Research Associates
- **Honeywell Software Development Project**, by *Lance Sherry*, Honeywell; and *Doug Hoover*, Odyssey Research Associates

# The AAMP5/AAMP–FV Project

Steven P. Miller
Collins Commercial Avionics
Rockwell International
Cedar Rapids, IA 52498 USA
spmiller@pobox.cca.rockwell.com

Mandayam Srivas
Computer Science Laboratory
SRI International
Menlo Park, CA 94025 USA
srivas@csl.sri.com

Software and digital hardware are increasingly being used in situations where failure could be life threatening, such as aircraft, nuclear power plants, weapon systems, and medical instrumentation. Several authors have demonstrated the infeasibility of showing that such systems meet ultra–high reliability requirements through testing alone [1,2]. Formal methods are a promising approach for increasing our confidence in digital systems, but many questions remain on how it can be used effectively in an industrial setting.

This presentation describes a project, formal verification of the microcode in the AAMP5 microprocessor, conducted to explore how formal techniques for specification and verification could be introduced into an industrial process. Sponsored by the Systems Validation Branch of NASA Langley and by Collins Commercial Avionics, a division of Rockwell International, it was conducted by Collins and the SRI International Computer Science Laboratory. The project consisted of specifying in the PVS language developed by SRI [3] a portion of a Rockwell proprietary microprocessor, the AAMP5, at both the instruction set and register–transfer levels and using the PVS theorem prover to prove the microcode correct for a representative subset of instructions.

While this presentation includes a brief technical overview (~~see [4,5] for a detailed~~ technical discussion), its emphasis is on the lessons learned in using PVS for an example of this size and the implications for using formal methods in an industrial setting. The central result of this project was to demonstrate the feasibility of formally specifying a commercial microprocessor and the use of mechanical proofs of correctness to verify microcode. This is particularly significant since the AAMP5 was not designed for formal verification, but to provide a more than three fold performance improvement, by pipelining instruction execution, while remaining object code compatible with the earlier AAMP2. As a consequence, the AAMP5 is one of the most complex microprocessors to which formal methods have been applied.

Another key result was the discovery of both actual and seeded errors. Two actual microcode errors were discovered and corrected during development of the formal specification, illustrating the value of simply creating a precise specification. Two seeded errors were systematically uncovered while doing correctness proofs. One of these was an actual error that had been discovered after first fabrication but left in the microcode provided to SRI. The other error was designed to be unlikely to be detected by walkthroughs, testing, or simulation.

Several other results emerged during the project, including the ease with which practicing engineers became comfortable with PVS, the need for libraries of general purpose theories, the usefulness of formal specification in revealing errors, the natural fit between formal specification and inspections, the difficulty of selecting the best style of specification for a new problem domain, the high level of assurance provided by proofs of correctness, and the need to engineer proof strategies for reuse.

Many of the costs of the AAMP5 project can be attributed to the overhead of applying an experimental method for the first time. To determine how much these costs can be reduced through reuse of the AAMP5 expertise, Collins, SRI, and NASA are conducting a follow–on project to verify the microcode in the AAMP–FV, a smaller microprocessor design similar to those actually used in autoland systems. A report on the status of this project is also presented.

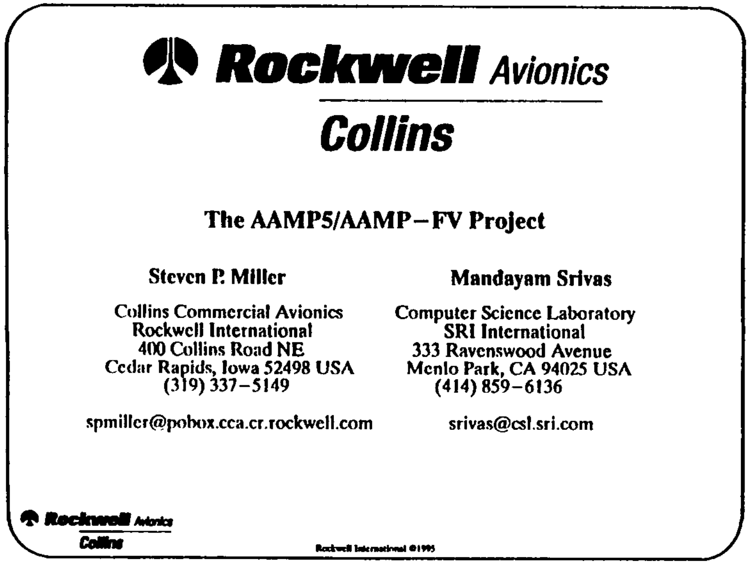
[1] Butler, R. and G. Finelli, The Infeasibility of Experimental Quantification of Life–Critical Software Reliability, *Software Engineering Notes*, Vol. 16, No.5, pg. 66–76, December 1991.

[2] Littlewood, B. and L. Strigini, Validation of Ultra–High Dependability for Software–based Systems, *Communications of the ACM*, Vol. 36, No. 11, pg. 69–80, November 1993.

[3] Owre, S., J. Rushby, and N. Shankar, PVS: A Prototype Verification System, In Deepak Kapur, Editor, *11th International Conference on Automated Deduction, (CADE)*, pg. 748–752, Saratoga, NY, June 1992, Vol. 607 of Lecture Notes in Artificial Intelligence, Springer–Verlag.

[4] Srivas, M. and S. Miller, Formal Verification of the AAMP5: A Case Study in the Verification of a Commercial Microprocessor, to appear in *Applications of Formal Methods*, Michael G. Hinchey and Jonathan P. Bowen, Editors, Prentice–Hall International Series in Computer Science.

[5] Srivas, M. and S. Miller, *Formal Verification of an Avionics Microprocessor,* to be submitted as a NASA Contractor Report.



Rockwell Avionics
Collins

## The AAMP5/AAMP–FV Project

Steven P. Miller

Collins Commercial Avionics
Rockwell International
400 Collins Road NE
Cedar Rapids, Iowa 52498 USA
(319) 337–5149

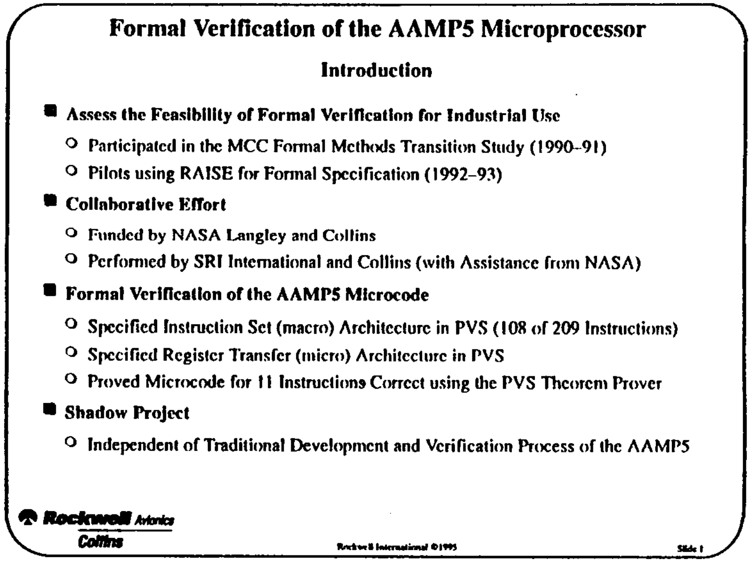
spmiller@pobox.cca.cr.rockwell.com

Mandayam Srivas

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA
(414) 859–6136

srivas@csl.sri.com

Rockwell Avionics Collins

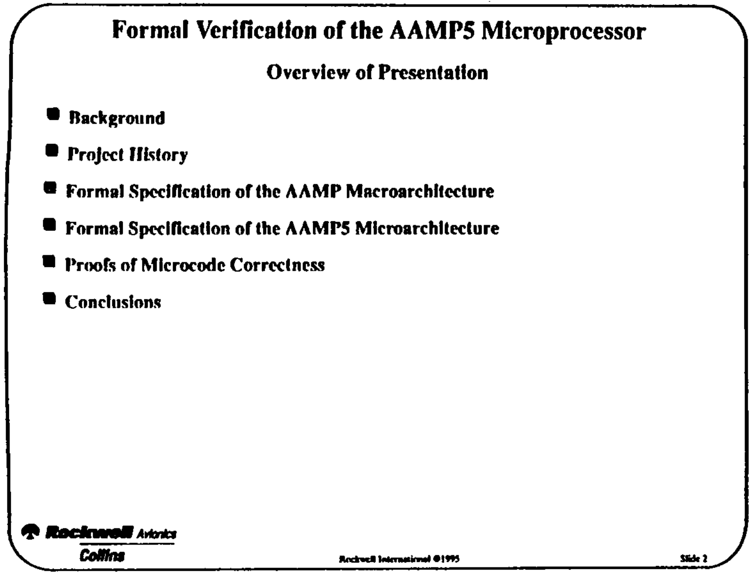


## Formal Verification of the AAMP5 Microprocessor

### Introduction

- Assess the Feasibility of Formal Verification for Industrial Use
  - ○ Participated in the MCC Formal Methods Transition Study (1990–91)
  - ○ Pilots using RAISE for Formal Specification (1992–93)
- Collaborative Effort
  - ○ Funded by NASA Langley and Collins
  - ○ Performed by SRI International and Collins (with Assistance from NASA)
- Formal Verification of the AAMP5 Microcode
  - ○ Specified Instruction Set (macro) Architecture in PVS (108 of 209 Instructions)
  - ○ Specified Register Transfer (micro) Architecture in PVS
  - ○ Proved Microcode for 11 Instructions Correct using the PVS Theorem Prover
- Shadow Project
  - ○ Independent of Traditional Development and Verification Process of the AAMP5

Rockwell Avionics Collins

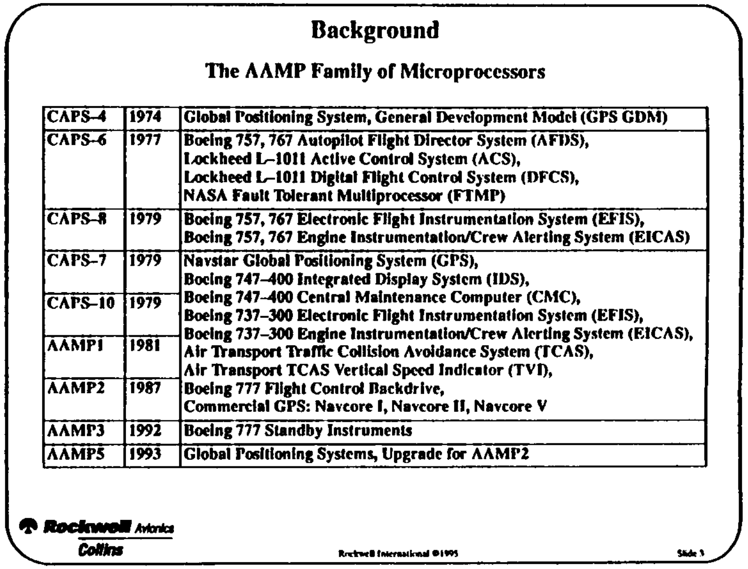


## Formal Verification of the AAMP5 Microprocessor

### Overview of Presentation

- Background
- Project History
- Formal Specification of the AAMP Macroarchitecture
- Formal Specification of the AAMP5 Microarchitecture
- Proofs of Microcode Correctness
- Conclusions

Rockwell Avionics Collins



## Background

### The AAMP Family of Microprocessors

| | | |
|---|---|---|
| CAPS-4 | 1974 | Global Positioning System, General Development Model (GPS GDM) |
| CAPS-6 | 1977 | Boeing 757, 767 Autopilot Flight Director System (AFDS),<br>Lockheed L-1011 Active Control System (ACS),<br>Lockheed L-1011 Digital Flight Control System (DFCS),<br>NASA Fault Tolerant Multiprocessor (FTMP) |
| CAPS-8 | 1979 | Boeing 757, 767 Electronic Flight Instrumentation System (EFIS),<br>Boeing 757, 767 Engine Instrumentation/Crew Alerting System (EICAS) |
| CAPS-7 | 1979 | Navstar Global Positioning System (GPS),<br>Boeing 747-400 Integrated Display System (IDS),<br>Boeing 747-400 Central Maintenance Computer (CMC),<br>Boeing 737-300 Electronic Flight Instrumentation System (EFIS),<br>Boeing 737-300 Engine Instrumentation/Crew Alerting System (EICAS),<br>Air Transport Traffic Collision Avoidance System (TCAS),<br>Air Transport TCAS Vertical Speed Indicator (TVI), |
| CAPS-10 | 1979 | |
| AAMP1 | 1981 | |
| AAMP2 | 1987 | Boeing 777 Flight Control Backdrive,<br>Commercial GPS: Navcore I, Navcore II, Navcore V |
| AAMP3 | 1992 | Boeing 777 Standby Instruments |
| AAMP5 | 1993 | Global Positioning Systems, Upgrade for AAMP2 |

Rockwell Avionics Collins

## Background

### AAMP Family of Microprocessors

- Based on Stack Architecture
- Designed for Use in Embedded Systems with High-Level Languages (Ada)
- Multiple Addressing Modes
- Large, CISC-like Instruction Set
- Double Code Density of Most Microprocessors
- Provides Direct Hardware Support for Much of
  - Run Time Environment
  - Real-Time Executive

Rockwell Avionics Collins

## Background

### AAMP5

- Object Code Compatible with Earlier AAMP2
- Uses Pipelining to Achieve 3x Performance of the AAMP2
- ~500,000 Transistors
  - ~20,000 Transistors in Other Formal Microprocessor Verification Efforts
  - ~3.1 Million Transistors in an Intel Pentium
- Performance Between Intel 386 and 486
- Intended for Use in Avionics Displays and Global Positioning Systems

Rockwell Avionics Collins

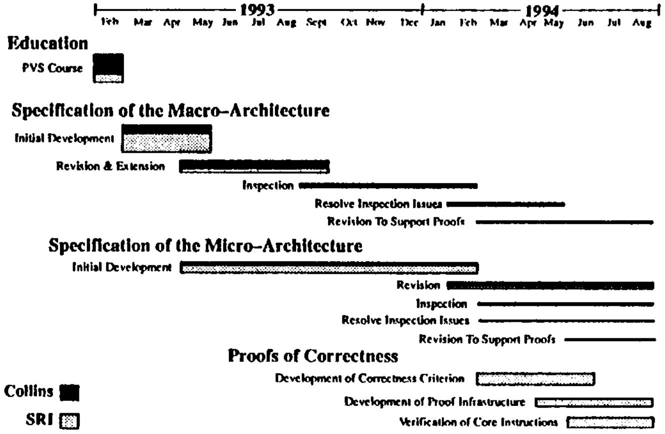
## Background

### PVS

- Environment for Formal Specification and Verification
- Developed by SRI International
- Expressive Specification Language
  - Simple and Easy to Learn
  - Notation Similar to Conventional Programming Languages
  - Sophisticated Type System
  - Higher Order Logic
- Typechecker
- Interactive Theorem Prover
  - Automates Many of the Low Level Proof Steps

Rockwell Avionics Collins

## Project History



Rockwell Avionics Collins

## Formal Specification of the AAMP Macroarchitecture

- **Initial Development Done by SRI**
  - 1,595 Lines of PVS and Comments Organized into 25 Theories
- **Bit Vectors Library Taken Over by NASA Langley**
  - Evolved into 2,030 Lines of PVS and Comments in 31 Theories
- **Revised and Extended by Collins**
  - Specified most of the Executive Service Routines
  - Completed 108 of the AAMP's 209 Instructions
  - 2,550 Lines of PVS and Comments Organized into 48 Theories
- **Performed 11 Inspections of the PVS Specifications**
  - Practicing Electrical Engineers
  - Inspectors had Little Trouble Reading PVS
  - Average Preparation Rate of ~150 Lines of PVS and Comments/Hour
  - Found 53 Minor (Style) Errors, 28 Major (Correctness) Errors

Rockwell Avionics Collins

Rockwell International ©1995

Slide 8

## Formal Specification of the AAMP Macroarchitecture

### Microcode Errors Found in This Phase

- **Discovered Two Microcode Errors**
  - Both Specific to the AAMP5
  - Both Corrected Before First Fabrication
  - Had Not Completed Traditional Verification Effort
- **Found While Specifying Behavior of the AAMP During Unusual Cases**
- **One was Due to a Missing Requirement**
  - Would Have Been Found During Ada Validation Testing
- **Other Was a Coding Error**
  - Required Specific Stack Configuration, an Improperly Sized Stack, and a Specific Instruction Sequence
- **Illustrates the Value of Simply Creating a Precise Specification**

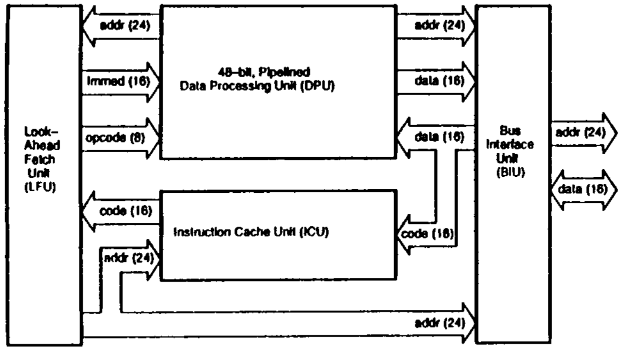
Rockwell Avionics Collins

Rockwell International ©1995

Slide 9

## Formal Specification of the AAMP5 Microarchitecture

### AAMP5 Block Diagram



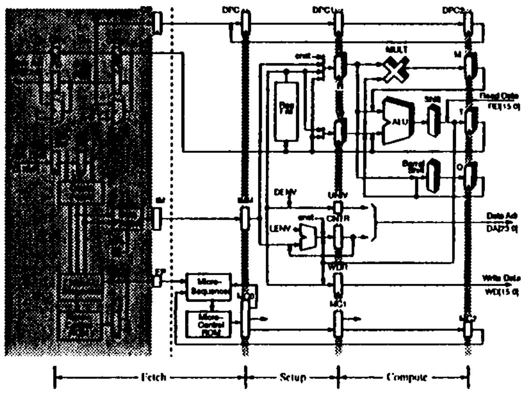
Rockwell Avionics Collins

Rockwell International ©1995

Slide 10

## Formal Specification of the AAMP5 Microarchitecture

### DPU Microarchitecture



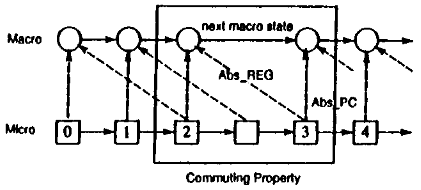
Rockwell Avionics Collins

Rockwell International ©1995

Slide 11

## Proofs of Microcode Correctness

### Approach and Issues



- Pipelining
- Autonomous Code and Data Access
- Enhancements to Support Efficient Automation
  - Boolean Decision Diagram (BDD) Decision Procedures
  - Optimizations to the PVS Rewriting Engine
- Decomposition of Verification Approach to Facilitate Technology Transfer

Rockwell Avionics Collins

Rockwell International ©1995

Slide 12

## Proofs of Microcode Correctness

### Microcode Errors Found in This Phase

- Seeded Two Errors in Microcode Provided to SRI
- First was Designed to be Difficult to Detect
- Second was an Actual Error
  - Escaped Detection by Simulations and Walkthroughs
  - Discovered by Collins While Running Application Code on an Early Prototype of the AAMP5
- Both Errors were Found by SRI While Doing the Proofs
  - Discovery was Very Systematic
  - SRI Explained How to Correct Both Errors

Rockwell Avionics Collins

Rockwell International ©1995

Slide 13

## Summary

- Specified 108 of the AAMP's 209 Instructions
- Specified the AAMP5 at the Register Transfer Level
- Developed a Large, Reuseable Library of Bit Vector Properties
- Discovered Two Errors in AAMP5 Microcode During Specification
- Conducted Inspections of the PVS Specifications With Design Engineers
- Developed General Approach to Decomposing the Proofs of Correctness
- Automated these Strategies in the PVS Prover
- Formally Verified the Microcode in 11 Instructions
- Discovered Two Seeded Errors during Formal Verification

Rockwell Avionics Collins

Rockwell International ©1995

Slide 14

## Conclusions

- Demonstrated the Technical Feasibility of
  - Formally Specifying the AAMP5 at Instruction Set and Register Transfer Levels
  - Formally Verifying the Microcode in the AAMP5
- Benefits of Formal Specification
  - Encourages Clean Abstractions and Interfaces
  - Encourages "Looking in Corners"
- PVS Specifications Successfully Used by Practicing Engineers
  - Synergy between Specifications and Inspections was Key
  - Acceptance of PVS by Engineers Varies Widely
  - Difficult to Enforce the Discipline Needed to Ensure Quality Specifications
- Could Achieve *Dramatic* Gains in Acceptance Through
  - Notations that Fit a Specific Problem Domain

Rockwell Avionics Collins

Rockwell International ©1995

Slide 15

## Conclusions

- **Formal Verification, Done Correctly, Provides Very High Levels of Assurance**
  - Does not Eliminate Good Process, Peer Reviews, Testing, Simulation, ...
  - May Facilitate or Lessen the Need for Some Traditional Practices
- **Expect Costs to be High the First Time in a New Problem Domain**
  - Expertise
  - Reusable Theories and Proofs
- **How Much Will Costs Drop on Subsequent Projects?**

Rockwell Avionics
Collins

Rockwell International ©1995

Slide 16

## AAMP-FV

- **Collaborative Effort**
  - Funded by NASA and Collins
  - Conducted by Collins and SRI
  - Initiated in January, 1995
- **Smaller Microprocessor**
  - Paper and Pencil Design
  - Similar to What We Would Use in an Autoland System
  - ~100,000 Transistors
- **Repeat AAMP5 Experiment**
  - Reuse Expertise and Theories
  - Demonstrate Cost Effectiveness
- **Current Status**
  - Specified Microarchitecture ( 50 Hours)
  - Nearly Completed the Proof of the First Instruction

Rockwell Avionics
Collins

Rockwell International ©1995

Slide 17

cont
to
p 109

*No slides are available*

*for*

*Joe Profeta's portion of this presentation*

# APPLYING FORMAL METHODS TO RAILWAY CONTROL

*Doug N. Hoover*

*Odyssey Research Associates, Inc.*

In collaboration with Union Switch and Signal, ORA has been carrying on research into applying formal methods to system-level railway control modeling and to verification of parts of VFRAME++, a railway control CAD system under development by US&S.

The railway modeling work has produced modeling methods powerful enough to prove safety of the conventional block control system. We hope to apply it to newer systems for which safety is more problematic.

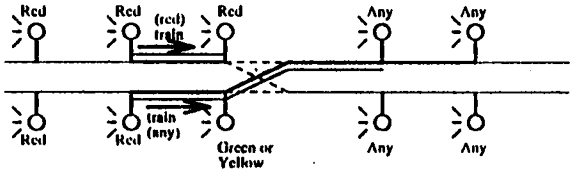
The VFRAME++ work centers around correctness of translation from graphical representations of circuits to hardware implementing them. Work so far carried out consists of design verification of a translation algorithm in PVS. Currently planned is an a posteriori checker to show that a particular translation has been done correctly. Such a checker would have the advantage of being simple and stand-alone, hence easy to demonstrate to be correct.

# Applying Formal Methods to Railway Control

*D. N. Hoover*
*Odyssey Research Associates, Inc.*
*301 Dates Dr.*
*Ithaca NY 14850-1326*
*Internet: hoove@oracorp.com*

## Outline of US&S/ORA Collaboration

- □ System-level railway control modeling.
- □ Verification of translations between different languages in VFRAME++ a railway control CAD system.
  - ○ Design verification of a translation algorithm.
  - ○ Automatic translation checker.



## Railway Control Safety Modeling

A ``hot" area.

- □ Most work is about verifying devices and software (e.g. SACEM–Paris-RER) used in automated control systems.
- □ Little system level model modeling, none for proof of safety (T.~King, K.M.~Hansen, FME '94 for specification, simulation).
- □ What we did:
  - ○ Developed railway modeling methods supporting mathematical proof of safety.
  - ○ Applied to classical block control system.
  - ○ Substantial parts formalized and basic properties proved in PVS in order to clarify definitions

## DAB-to-GEM Translation

Translation between graphical interface language (DAB) and generic intermediate language (GEM) in VFRAME++.

For exploratory purposes, we formalized the languages, their semantics, and a translation algorithm. We verified that the translation is correct (preserves semantics).

## DAB-to-GEM Translation

- □ DAB (Design Application Builder) representation — assignment complex Boolean expressions to output variables.

  w1 := u1 AND (u2 OR u3)
  w2 := (NOT u2) AND (TRUE OR u3)

  (Full language would also have some arithmetic and control structures.)
- □ GEM (Generic Entity Model) representation — simple assignments only.

## DAB-to-GEM Translation

| assignments | | assertion |
|---|---|---|
| v1 := u2 | % | v1 = u2 |
| v2 := NOT v1 | % | v2 = NOT u2 |
| v3 := TRUE | % | v3 = TRUE |
| v4 := u3 | % | v4 = u3 |
| v5 := v3 OR v4 | % | v5 = TRUE OR u3 |
| v6 := v2 AND v5 | % | v6 = (NOT u2) AND (TRUE OR u3) |
| v7 := u1 | % | v7 = u1 |
| v8 := v1 OR v4 | % | v8 = u2 OR u3 |
| v9 := v7 AND v8 | % | v9 = u1 AND (u2 OR u3) |
| w2 := v6 | % | w2 = (NOT u2) AND (TRUE OR u3) |
| w1 := v9 | % | w1 = u1 AND (u2 OR u3) |

## Results

- □ Some important basic concepts.
  - ○ How to define the expressions of a language with a grammar.
  - ○ The idea that expressions have a meaning (semantics).
  - ○ That a correct translation is one that preserves semantics.

$$\begin{tikzcd}
L \arrow[dr, "sem"] \arrow[dd, "tr"'] & \\
 & D \\
L' \arrow[ur, "sem'"'] &
\end{tikzcd}$$

- □ FM allows fancier programming without added risk.
  - ○ Our translation algorithm was more sophisticated than US\&S's.
  - ○ They were being cautious, we were trying to show off.

## Continuing Work

- □ Also of concern to US&S: Correct algorithm incorrectly coded.
  - ○ Correct source code incorrectly compiled.
  - ○ Correct object code producing wrong result due to a fault.
  - ○ How do you know the theorem prover/program verifier works correctly? (Tool qualification problem.)
- □ Therefore US&S prefer also to check a posterior that translation hasbeen done correctly.
- □ Advantage: a posteriori check can be done automatically.

## A posteriori checking

- ▢ A simple checker checks translation and produces a log.
- ▢ A really simple checker checks correctness of the log.
- ▢ Need to justify only the correctness of the log checker.
- ▢ Note: the a posteriori check seems to be simpler than the translation in this case.

## Example

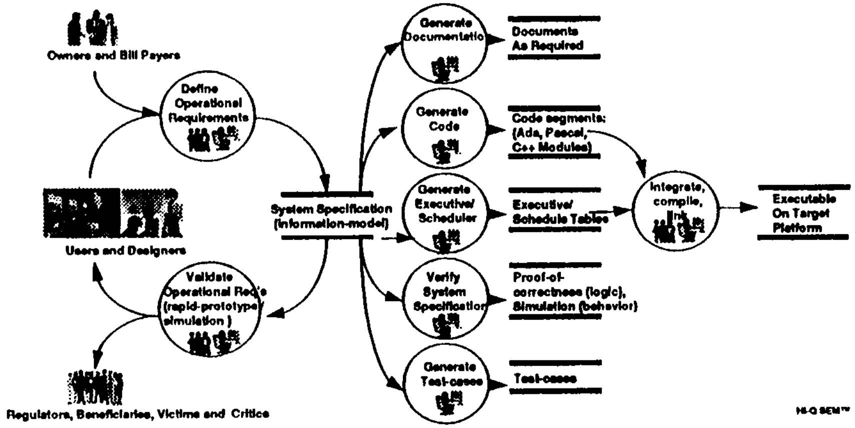
```
    w1 := u1 AND (u2 OR u3)
    w2 := (NOT u2) AND (TRUE OR u3)
assignments          assertion
 v1 := u2          %  v1 = u2
 v2 := NOT v1      %  v2 = NOT u2
 v3 := TRUE        %  v3 = TRUE
 v4 := u3          %  v4 = u3
 v5 := v3 OR v4    %  v5 = TRUE OR u3
 v6 := v2 AND v5   %  v6 = (NOT u2) AND (TRUE OR u3)
 v7 := u1          %  v7 = u1
 v8 := v1 OR v4    %  v8 = u2 OR u3
 v9 := v7 AND v8   %  v9 = u1 AND (u2 OR u3)
 w2 := v6          %  w2 = (NOT u2) AND (TRUE OR u3)
 w1 := v9          %  w1 = u1 AND (u2 OR u3)
```

## Conclusions

Value of FM

- ▢ Concepts from TCS—languages, semantics, abstraction, black-box-specification (precondition/postcondition). Not just algorithms, but-properties and states.
- ▢ Logical methods for manipulation and precise checking of semanticcontent of algorithms, code, results. A means of testing specifications.
- ▢ Education: concepts first, formalization later, or else avoid it altogether.
- ▢ In FM applications, often something simple is the most useful.

## III Model-Based System Development

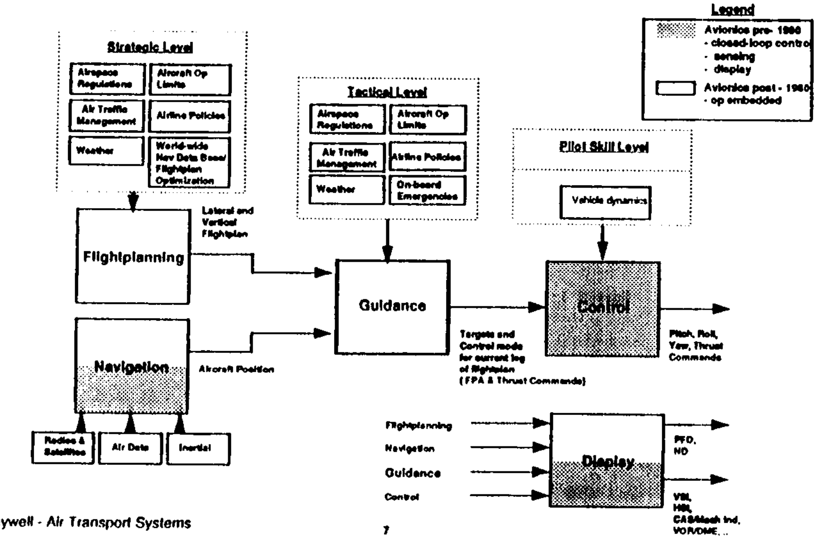

Owners and Bill Payers
Define Operational Requirements
Users and Designers
Validate Operational Req's (rapid-prototype/ simulation )
Regulators, Beneficiaries, Victims and Critics
System Specification (information-model)
Generate Documentatio
Documents As Required
Generate Code
Code segments: (Ada, Pascal, C++ Modules)
Generate Executive/ Scheduler
Executive/ Schedule Tables
Verify System Specificatio
Proof-of-correctness (logic), Simulation (behavior)
Generate Test-cases
Test-cases
Integrate, compile, link
Executable On Target Platform
HI-Q SEM™

## IV Characteristics Of The Avionics Information-Model

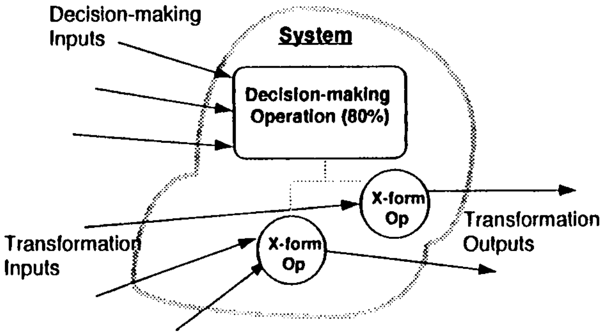
- Capture *operationally embedded* nature of systems
- Capture *reactive* nature of systems
- Provide means for rapid formulation and adaptation of "*mental-models*" of the system
- Provide means for rapid formulation and adaptation of "*implementation-models*" of the system

(see definitions next three pages)

## IV Avionics Info-Models: Operationally Embedded Systems



## IV Avionics Info-Models: Reactive Systems



## Model-based Development of Software-based Systems:

## A Research Agenda

May 1994

| NASA-Langley | ORA | Honeywell |
|---|---|---|
| Michael Holloway | Doug Hoover | Lance Sherry |
| | Zwei Chen | Dave Youssefi |
| | David Guaspari | Matt Michaels |
| | | Christine Mixon |
| | | Jack Janelle (and team) |
| | | Dave Cirulli |
| | | Joel Thornton (and team) |
| | | Jon Ward |
| | | John Krueger . . . |

## Table Of Contents

I Overview: Integrated Cockpit Avionics

II Documentation-based Process

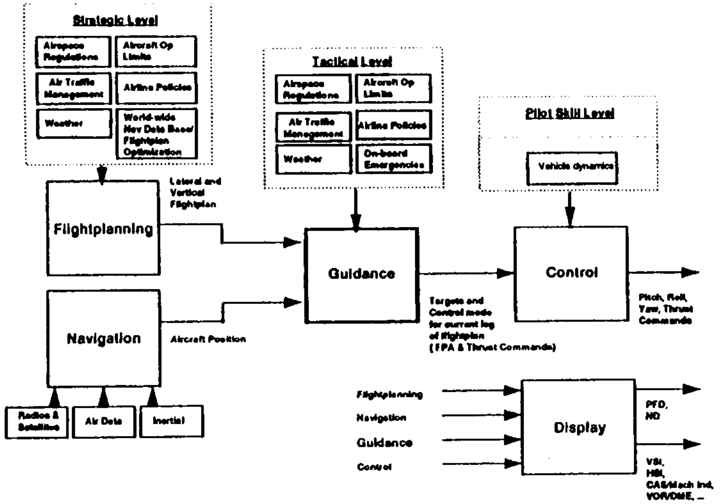
III Model-based Process

IV Characteristics Of The Avionics Information-Model

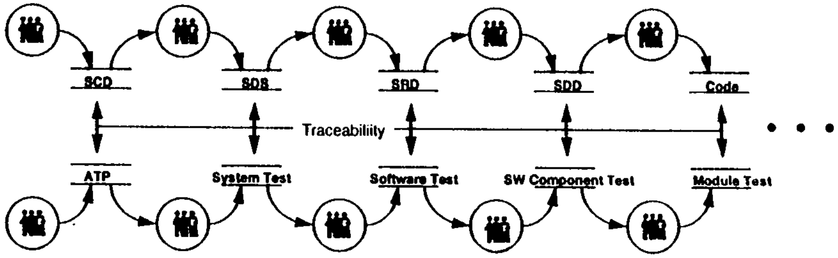
V Current Research

VI Summary/Roadmap

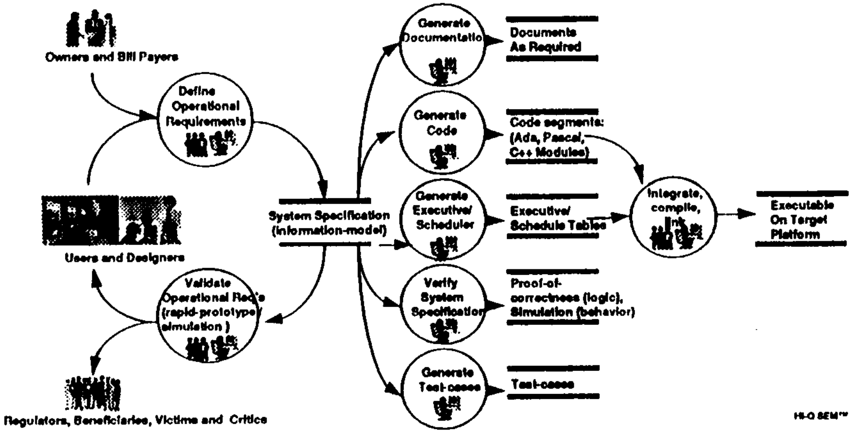
## I Overview: Integrated Cockpit Avionics



## II Documentation-Based System Development



## III Model-Based System Development

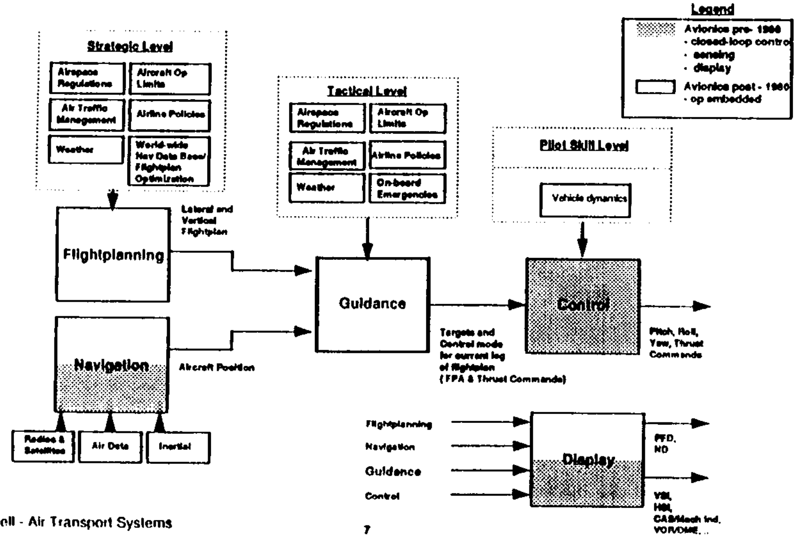

Owners and Bill Payers
Define Operational Requirements
Users and Designers
Validate Operational Req's (rapid-prototype/ simulation )
Regulators, Beneficiaries, Victims and Critics
System Specification (information-model)
Generate Documentatio
Documents As Required
Generate Code
Code segments: (Ada, Pascal, C++ Modules)
Generate Executive/ Scheduler
Executive/ Schedule Tables
Verify System Specificatio
Proof-of-correctness (logic), Simulation (behavior)
Generate Test-cases
Test-cases
Integrate, compile, link
Executable On Target Platform
HI-Q SEM™

## IV Characteristics Of The Avionics Information-Model

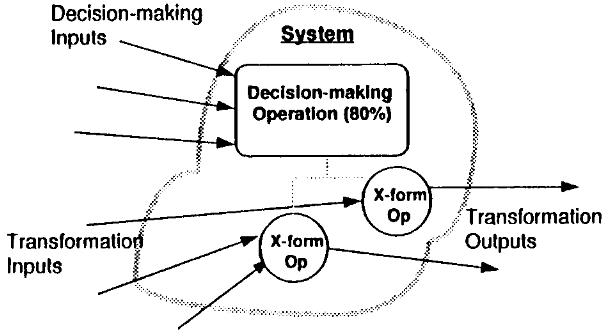
- Capture *operationally embedded* nature of systems
- Capture *reactive* nature of systems
- Provide means for rapid formulation and adaptation of "*mental-models*" of the system
- Provide means for rapid formulation and adaptation of "*implementation-models*" of the system

(see definitions next three pages)

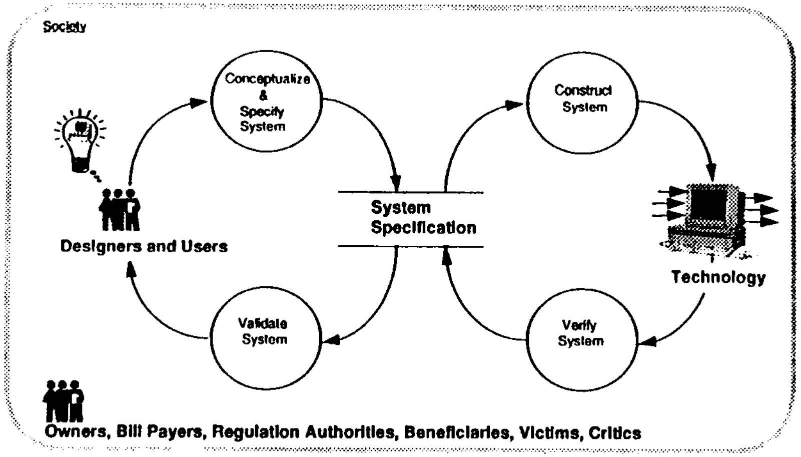
## IV Avionics Info-Models: Operationally Embedded Systems



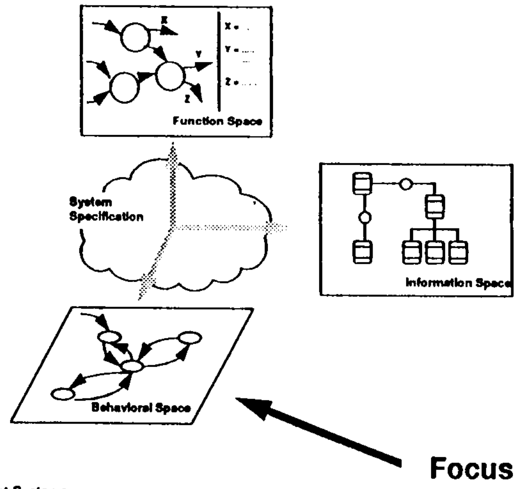
## IV Avionics Info-Models: Reactive Systems



## IV Avionics Info-Models: Rapid Formulation and Adaptation of Mental-models and Implementation-models

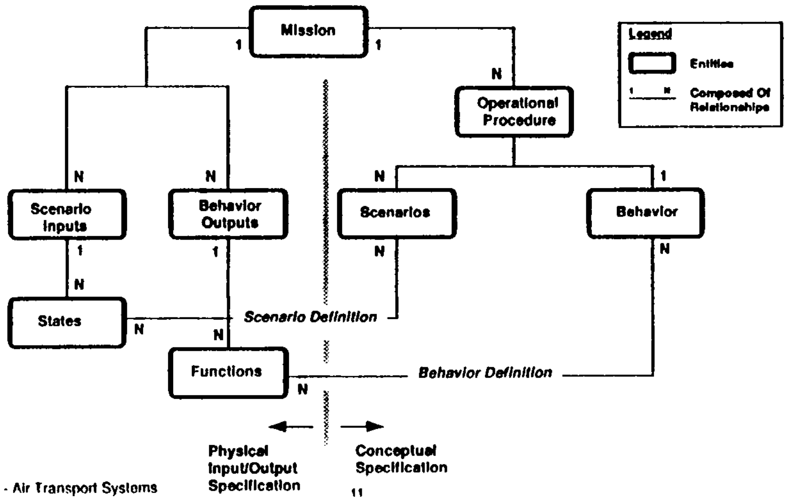

Society
Conceptualize & Specify System
Construct System
System Specification
Designers and Users
Technology
Validate System
Verify System
Owners, Bill Payers, Regulation Authorities, Beneficiaries, Victims, Critics

## V Current Research



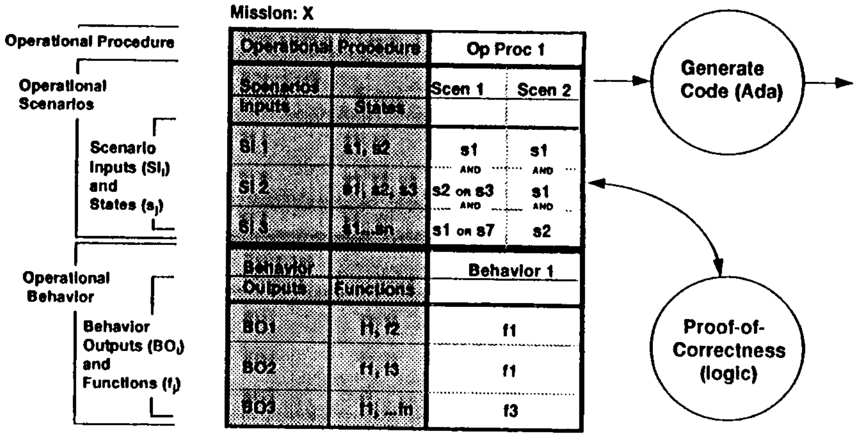
## V Current Research

### Operational Procedure Information Model



## V Current Research

### Visualization Of Info-Model and Automation


Operational Procedure
Operational Scenarios
Scenario Inputs (SI$_i$) and States (s$_j$)
Operational Behavior
Behavior Outputs (BO$_i$) and Functions (f$_j$)
Generate Code (Ada)
Proof-of-Correctness (logic)

Mission: X

| Operational Procedure | | Op Proc 1 | |
|---|---|---|---|
| Scenarios | | Scen 1 | Scen 2 |
| Inputs | States | | |
| SI 1 | s1, s2 | s1 AND | s1 AND |
| SI 2 | s1, s2, s3 | s2 OR s3 AND | s1 AND |
| SI 3 | s1...sn | s1 OR s7 | s2 |
| Behavior | | Behavior 1 | |
| Outputs | Functions | | |
| BO1 | f1, f2 | f1 | |
| BO2 | f1, f3 | f1 | |
| BO3 | f1, ...fn | f3 | |

## VI Summary

- Economic pressures driving change in industry (reduce cost, reduce cycle-time, improve quality)
- Paradigm shift from documentation-based to model-based development process
- Model-based development process requires definition of "Information-model"
- Characteristics Of Avionics Info-model:
  - *operationally embedded*
  - *reactive*
  - easy-to-read for "*mental-modelling*"
  - easy-to-code for "*implementation-modelling*"

## VI Roadmap

| | |
|---|---|
| 1995 | Model-based development (single user) |
| 1996 | Model-based development (multi-user)<br>- shared specification, dictionaries, Config Management |
| 1997 | Model-based development (multi-user) |
| 1998 | Model-based development (multi-user)<br>- includes customers and vendors |
| 1999 | Certification credit for automated analysis/verification |

# FORMAL TOOLS AND METHODS FOR DECISION TABLES

*Doug N. Hoover*

*Odyssey Research Associates, Inc*

This work began with a problem from Honeywell: how to tell whether a specification or code involving a complex choice of alternatives is complete (always designates a choice) and is consistent (always designates only one choice). It turned out that Honeywell used decision tables informally to specify this kind of code. Now, decision tables are a kind of formal specification, so we decided that the best solution was to build a specialized tool, TableWise, to deal with the problem.

TableWise checks completeness and consistency of decision tables, as well as generating Ada code and documentation from them. It has an original form of structural analysis that localizes flaws that prevent decision tables from being complete and consistent. TableWise is available from NASA Langley by anonymous ftp (air16.larc.nasa.gov). A paper on TableWise will appear in COMPASS '95.

Continuing work related to TableWise includes generating tests from decision tables, improving code generation, structuring decision tables to compress information, and making decision tables part of a more all-encompassing specification methodologies.

# Formal Tools and Methods for Decision Tables

*D. N. Hoover*
*Odyssey Research Associates, Inc.*
*301 Dates Dr.*
*Ithaca NY 14850-1326*
*Internet: hoove@oracorp.com*

## Overview

Honeywell's problem:

- ❐ Code involving choice of alternatives depending on complex conditions.
- ❐ Is the choice always unambiguous? (consistency)
- ❐ Is there always a choice? (completeness)

Raw material:

- ❐ Some Honeywell specs and code.
- ❐ Specs "nearly formal."
- ❐ Logical problem "nearly propositional" (some numerical comparisons).
- ❐ "The specs come from decision tables."
- ❐ Decision tables *are!* formal specs.

## Decision Tables

| Operational Procedure | | Takeoff | | Climb | | Climb_<br>Int_<br>Level | Cruise |
|---|---|---|---|---|---|---|---|
| Input Variables | States | | | | | | |
| Flightphase | climb<br>cruise | climb | climb | climb | climb | climb | cruise |
| AC_Alt > 400 | TRUE<br>FALSE | TRUE | TRUE | • | • | • | • |
| compare(AC_Alt, ACC-ALT) | LT EQ<br>GT | LT | LT | EQ GT | EQ GT | • | GT |
| Alt_Capt_Hold | TRUE<br>FALSE | FALSE | TRUE | FALSE | TRUE | TRUE | TRUE |
| compare(Alt_Target, prev_Alt-Target | LT EQ<br>GT | • | GT | • | GT | • | EQ |

## TableWise, a Decision Table Tool

After preliminary experimentation with PVS, we decided to build a specialized tool for the problem.

- ❐ Check completeness and consistency of decision tables
- ❐ Generate Ada code
- ❐ Generate English language specs like Honeywell's.
- ❐ Best of all, it is available by anonymous ftp from

```
air16.larc.nasa.gov
```

directory

```
fm/ora
```

## TableWise—Internals

- Logic by D. N. Hoover.
- Interface by Zewei Chen.
- The logic is in the functional subset of SML.
- **That constitutes a formal design spec.**
- Code generation tested by verifying the generated code using ORA's Penelope Ada verification system.
- BDD-type algorithms.

## Analysis Results

| Overlapping scenarios | | Takeoff.2<br>Climb_Int_Level.1 | Climb.2<br>Climb_Int_Level.1 |
|---|---|---|---|
| Input Variables | States | | |
| Flightphase | climb<br>cruise | climb | climb |
| AC_Alt > 400 | TRUE<br>FALSE | TRUE | • |
| compare(AC_Alt, ACC-ALT) | LT EQ<br>GT | LT | EQ GT |
| Alt_Capt_Hold | TRUE<br>FALSE | TRUE | TRUE |
| compare(Alt_Target, prev_Alt-Target | LT EQ<br>GT | GT | GT |

## Analysis Results (cont'd)

| Missing scenarios | | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|
| Input Variables | States | | | | |
| Flightphase | climb<br>cruise | climb | cruise | cruise | cruise |
| AC_Alt > 400 | TRUE<br>FALSE | TRUE | • | • | • |
| compare(AC_Alt, ACC-ALT) | LT EQ<br>GT | LT | LT EQ | GT | GT |
| Alt_Capt_Hold | TRUE<br>FALSE | FALSE | • | TRUE | FALSE |
| compare(Alt_Target, prev_Alt-Target | LT EQ<br>GT | • | • | LT GT | • |

## Structural Analysis

A way to localize flaws. There is always a structural flaw in an incomplete table.

| Operational Procedure | | Takeoff | | Climb | | Climb_Int_Level | Cruise |
|---|---|---|---|---|---|---|---|
| Input Variables | States | | | | | | |
| Flightphase | climb<br>cruise | climb | climb | climb | climb | climb | cruise |
| AC_Alt > 400 | TRUE<br>FALSE | TRUE | TRUE | • | • | • | • |
| compare(AC_Alt, ACC-ALT) | LT EQ<br>GT | LT | LT | EQ GT | EQ GT | • | GT |
| Alt_Capt_Hold | TRUE<br>FALSE | FALSE | TRUE | FALSE | TRUE | TRUE | TRUE |
| compare(Alt_Target, prev_Alt-Target | LT EQ<br>GT | • | GT | • | GT | • | EQ |

## Structu

| 5 variable_structural flaws | | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|
| Input Variables | States | | | | |
| Flightphase | climb<br>cruise | • | • | cruise | cruise |
| AC_Alt > 400 | TRUE<br>FALSE | FALSE | • | • | • |
| compare(AC_Alt, ACC-ALT) | LT EQ<br>GT | • | • | LT EQ | • |
| Alt_Capt_Hold | TRUE<br>FALSE | • | • | • | FALSE |
| compare(Alt_Target, prev_Alt-Target | LT EQ<br>GT | • | LT | • | • |

This idea is new, based on "Hoover's Theorem 'B'."

## Continuing Work

- □ Assertions about decision tables.
- □ Decision tables including "behaviors," i.e. code implementing the action chosen.
- □ Generating tests from decision tables.
- □ Better code generation.
- □ Generating traceability information.
- □ Incorporate some of Parnas's ideas about tabular specifications.
- □ Tables as specification of a state transition (cf. Leveson's RSML).
- □ Structuring decision tables to compress information.
- □ Incorporate support for linear arithmetic in the checking.
- □ Solve the "industrial inference problem." (See Parnas, "Some Theorems We should Prove."

# Session 3: Industry Perspectives on Formal Methods

## C. Michael Holloway, Chair

- *Scott French*, Loral ATC (*No slides available*)
- *Steve Miller*, Rockwell-Collins
- *Joe Profeta*, Union Switch & Signal (*No slides available*)
- *Lance Sherry*, Honeywell

Rockwell Avionics

Collins

## Panel Discussion:
## Industry Perspectives on Formal Methods

Dr. Steven P. Miller

Advanced Technology & Engineering
Collins Commercial Avionics, MS 124-211
400 Collins Road NE
Cedar Rapids, Iowa 52498
(319) 395-8008

spmiller@pobox.cca.rockwell.com

## Who We Are

- Rockwell International
  - 75,000 Employees
  - Approximately 70% Sales are Commercial (Non-Government)
- Collins Commercial Avionics (CCA)
  - Cedar Rapids, Iowa
  - Civil Air Transport (Boeing, Lockheed, Fokker)
  - Civil General Aviation (Canadair, Beech, Cessna, Falcon, Embraer)
- Collins Communication and Avionics Division (CACD)
  - Cedar Rapids, Iowa
  - Defense and Government Systems (DOD and NASA)

## What We Do

- Collins Commercial Avionics (CCA)
  - Autopilot and Autoland Systems
  - Avionics Displays (EFIS, EICAS, PFD)
  - Traffic Alert and Collision Avoidance Systems (TCAS)
  - Communications Systems (Satellite)
  - Satellite Navigation Systems
- Collins Communication and Avionics Division (CACD)
  - Global Positioning Systems (GPS)
  - Joint Tactical Information Display System (JTIDS)
  - Mission Management Cockpit Display Systems (F-22)
  - Communication Systems

## Requirements Analysis

### Needs & Opportunities

- **Better Methods for Requirements Analysis is Our Single Greatest Need**
- ***Requirements are incomplete, misunderstood, poorly defined, and change in ways that are difficult to manage*** **– Software Productivity Consortium**
- **A Methodology for Requirements Analysis Must**
  - Be Readable by Domain Experts and Our Customers
  - Accommodate Frequent Change
  - Discourage Overspecification
  - Be Precise Enough to Support Formal Analysis and Development of Tools
  - Support Generation of Test Cases
  - Support Traceability
  - Tell You When You're Done

## Requirements Analysis

### State of Formal Methods

- "Traditional" Formal Methods Lack Methods
  - User Must Select Appropriate Approach for Each Problem Domain
  - Few "How To" Books
  - Success Very Dependent on the Skill of the User
- There are a Few Formal or Semi–Formal Methods Emerging
  - SCR – Software Cost Reduction Method (NRL)
  - CoRE – Consortium Requirements Engineering Method (SPC)
  - RSML – Requirements State Machine Language (Irvine)
- Underlying Formalism is Still Evolving
- Need More Tools

## Requirements Analysis

### What Should NASA Do?

- Establish Program in Requirements Engineering
- Collaborate with Other Groups Working on Requirements Engineering
  - Help to Complete the Formalization of Models
  - Support Development of Tools
- Emphasize Industry/Academic/Government Collaboration
  - Use Pilot Projects to Validate Methods and Tools

## Software Testing

### Needs & Opportunities

- Software Testing is Our Single Most Costly Activity
- Structural Unit Testing of Software on a Level A (Highest Criticality) System
  - Can Consume Over 50% of Total Project Budget (Hardware & Software)
  - Yet Finds Remarkably Few Errors
- Rapidly Moving Towards Requirements Based Testing
  - Create Tests from Requirements and Design Specifications
  - Determine Structural Coverage of Code as a Byproduct
- Needs
  - Execute Tests Against the Specifications
    - *Validate Requirements and Designs*
    - *Facilitate Early Development of Test Suites*
  - Automatically Generate Tests from Specifications

## Software Testing

### State of Formal Methods

- Need Methods for Formally Specifying Requirements and Designs
- Need Tools to Animate Specifications
- Need Methods for Generating Tests from Formal Specifications
  - Will Test Generation Differ for Each Specification Method?
  - What Distinguishes a "Useful" Test?

## Software Testing

### What Should NASA Do?

- Establish Program in Requirements Engineering
- Collaborate with Other Groups Working on Requirements Engineering
  - Explore Methods for Generation of Tests
  - Support Development of Tools
- Emphasize Industry/Academic/Government Collaboration
  - Use Pilot Projects to Validate Methods and Tools

## Logical Partitioning

### Needs & Opportunities

- Isolation of Critical from Non-Critical Functions On a Single Processing Site
  - Guarantee a Non-Critical Function Cannot Interfere with a Critical Function
  - Currently Provided by *Federated Architectures* Found in Most Civil Aircraft
- Proposed as a Mechanism to Reduce Costs and Improve Reliability
  - Reduce Flight Deck Space, Power, and Interconnect
  - Improve Reliability Through Use of Common Processing and I/O Units
- Also Essential for Coping with Increasing Demands for Functionality
  - Partition System in Critical and Non-Critical Functions
  - Focus Verification on Critical Functions
- Partitioning Mechanism Becomes the Most Critical Component

## Logical Partitioning

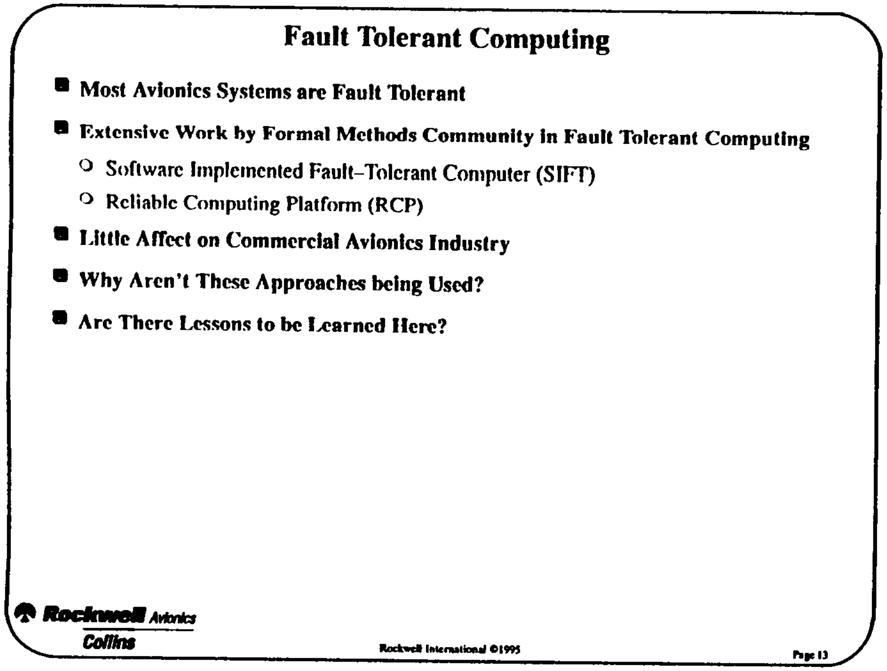
### State of Formal Methods

- Natural Application for "Traditional" Formal Methods
  - Fault-Tolerant Designs
  - Complex Mixture of Hardware and Software
  - Requires Extremely High Levels of Assurance
- Little Work by the Formal Methods Community

## Logical Partitioning

### What Should NASA Do?

- Lead Application of Formal Methods to Logical Partitioning
  - Rigorously Define the Basic Requirements of Logical Partitioning
  - Explore a Variety of Design Alternatives
  - Formally Verify the Correctness of Key Algorithms and Designs
- Emphasize Industry/Academic/Government Collaboration
  - Support Joint Efforts to Ensure Right Problems are Explored



## Fault Tolerant Computing

- Most Avionics Systems are Fault Tolerant
- Extensive Work by Formal Methods Community in Fault Tolerant Computing
  - ○ Software Implemented Fault-Tolerant Computer (SIFT)
  - ○ Reliable Computing Platform (RCP)
- Little Affect on Commercial Avionics Industry
- Why Aren't These Approaches being Used?
- Are There Lessons to be Learned Here?

Rockwell Avionics Collins

Q1) Company Needs From FM / Opportunities ?

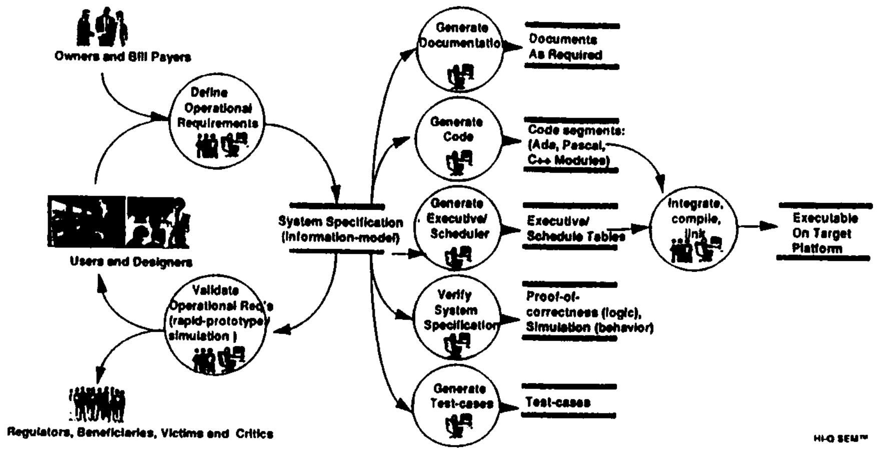
A1) Improve system safety.

Reduce development costs and cycle-time.

Support Model-based Avionic System Development

- capture operational embedded nature of systems
- capture reactive nature of systems
- easy to understand (visualization)
- easy to generate code

## Model-Based System Development


Owners and Bill Payers
Define Operational Requirements
Users and Designers
Validate Operational Req's (rapid-prototype simulation )
Regulators, Beneficiaries, Victims and Critics
System Specification (information-model)
Generate Documentatio
Documents As Required
Generate Code
Code segments: (Ada, Pascal, C++ Modules)
Generate Executive/ Scheduler
Executive/ Schedule Tables
Verify System Specificatio
Proof-of-correctness (logic), Simulation (behavior)
Generate Test-cases
Test-cases
Integrate, compile, link
Executable On Target Platform
HI-Q SEM™

Q2) State of FM ?

A2) Concept strong.

Areas Requiring Additional Work:

- Strengthen visualization techniques.
- Take advantage of code generation .
- Address regulatory issues.

Q3) Transfer of technology ?

A3)
- Educate, Educate, Educate (be patient)
- Package and market easy-to-use product
- Address cycle-time and cost issues (as well as safety)
- Demonstrate development & certification of "real avionic system" (joint FAA/NASA/ORA/Honeywell pilot project)

# Session 4: Software Systems (1)

**Ricky W. Butler, Chair**

- **Formal Verification for Fault-Tolerant Architectures/PVS Design**, by *John M. Rushby*, SRI International

- **Formal Methods Demonstration Project for Space Applications**, by *John Kelly*, Jet Propulsion Laboratory; and *Ben DiVito*, VÍGYAN, Inc.

# FAULT-TOLERANT ALGORITHMS AND THE DESIGN OF PVS

*John Rushby*

*SRI International*

PVS is the most recent in a series of verification systems developed at SRI. Its design was strongly influenced, and later refined, by our experiences in developing formal specifications and mechanically checked verifications for the fault-tolerant architecture, algorithms, and implementations of a model "reliable computing platform" (RCP) for life-critical digital flight-control applications.

Several of the formal specifications and verifications performed in support of RCP are individually of considerable complexity and difficulty. But in order to contribute to the overall goal, it has often been necessary to modify completed verifications to accomodate changed assumptions or requirements, and people other than the original developer have often needed to understand, review, build on, modify, or extract part of an intricate verification.

In this talk, I will outline the verifications performed, present the lessons learned, and describe some of the design decisions taken in PVS to better support these large, difficult, iterative, and collaborative verifications.

The following article covers this material in more detail:

"Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS" by Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke (*IEEE Transactions on Software Engineering*, Vol 21., No. 2, pp. 107-125).

This article is available on the World-Wide Web at the following URL:

http://www.csl.sri.com/tse95.html

# Fault-Tolerant Algorithms and the Design of PVS*

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

*PVS was built by Sam Owre and N. Shankar, with help from David Cyrluk, Pat Lincoln, Steven Phillips, Sreeranga Rajan, and Carl Witty. Its development was funded by SRI International

## Overview

- Formal methods applied to fault-tolerant algorithms and redundancy management for digital flight control
- Lessons learned about how mechanized formal methods can contribute to systems engineering and analysis: there's more to it than just proving correctness
- Lessons learned about requirements for tools
- And how these influenced the design of PVS

## Perspective

- Formal methods contribute useful mental frameworks, notations, and systematic methods to the design, documentation, and analysis of computer systems
- But the primary benefit from specifically *formal* methods is that they allow certain questions about a design to be answered by calculation (specifically, formal deduction)
- Thus some activities that are traditionally performed by *reviews*
  - o Processes that depend on human judgment and consensus

  can be replaced or supplemented by *analyses*
  - o Processes that can be repeated and checked by others, and potentially so by machine

(Terminology from RTCA DO-178B/EUROCAE ED-12B)

## Mechanization of Formal Methods

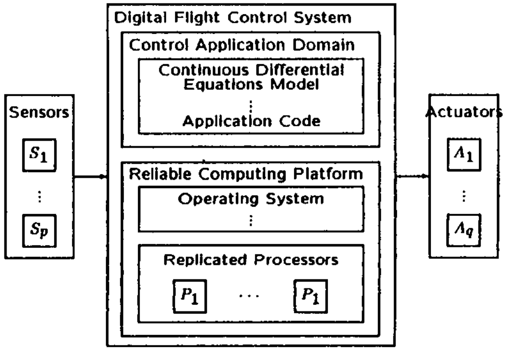
- Just like any other engineering calculations, it's tools that make formal calculations feasible in practice
- Tools are not the most important thing about formal methods
  - o They are the *only* important thing
- Building tools is an engineering enterprise
- The most important attribute of a tool is its speed and power
- If it cannot deal with problems of practical interest at reasonable cost, doesn't matter what other attributes it has

## Context

**Goal:** develop a formally verified architecture, algorithms, and implementations for a reliable computing platform for digital flight control based on state machine replication

**Reliable Computing Platform:** a distributed collection of processors, with associated sensors and actuators, and an "operating system" providing fault tolerance and redundancy management so that to an application task, it all looks like a single computer that never fails

## Reliable Computing Platform


Sensors
$S_1$
⋮
$S_p$
Digital Flight Control System
Control Application Domain
Continuous Differential Equations Model
⋮
Application Code
Reliable Computing Platform
Operating System
⋮
Replicated Processors
$P_1$
…
$P_1$
Actuators
$A_1$
⋮
$A_q$

## Context (ctd. 1)

**Formally Verified:** Construct a mathematical model of the system and its environment and prove a theorem of the form

**if** certain assumptions are true
**and** there are enough nonfaulty processors
**then** overall system masks all faults

**Assumptions:** must be validated in the real world (e.g., nonfaulty processors can read each other's clocks with error at most $\epsilon$)

**Enough nonfaulty processors:** must collect empirical data on individual failure rates, then use (Markov) modeling

## Context (ctd. 2)

**State machine replication:**

- Keep processors synchronized $\surd$
- Distribute identical sensor data to all processors $\surd$
- Run identical workloads on all processors
- Perform exact-match majority voting on values sent to actuators (provides fault masking) $\surd$
- Incrementally refresh state of all processors with majority-voted versions (provides transient recovery) $\surd$
- Optionally, diagnose faulty processors and reconfigure (increases number of faults that can be tolerated) $\surd$

## Context (ctd. 3)

**Why do this?**

Because redundancy management is among the most difficult problems in digital flight control

- Historically the *primary* source of system failure

**Why state machine replication?**

Because it has a rational justification

And because it can withstand arbitrary (aka. Byzantine) faults

**Why does that matter?**

Because then you don't have to provide evidence that faults outside your assumed class cannot occur

**Why do formal verification if system is fault tolerant?**

Because it's fault tolerant wrt. *hardware* faults; the algorithms and implementations of fault tolerance are single points of failure: they must be correct

## Formal Verifications Performed: Clock Synchronization

Interactive Convergence Byzantine-fault-tolerant clock synchronization algorithm

- At the time (1988), one of the hardest computer-science verifications completed
- Over 200 lemmas (small theorems), 1700 lines of specification and proof commands (in EHDM)
- Found that in the journal presentation by Lamport and Melliar Smith (JACM), proof of the main theorem is flawed and 4 of the 5 lemmas are false
- Corrections required adjustment to assumptions
  - Also eliminated approximations
  - And developed more uniform proofs

## Clock Synchronization (ctd. 1)

Our proof was subsequently repeated by Bill Young using Boyer-Moore prover

- He noted that our characterization of a good clock

$$|c(T_1) - c(T_2) - (T_1 - T_2)| < \frac{\rho}{2}|T_1 - T_2|$$

is unsatisfiable when $T_1 = T_2$, and also excludes perfect clocks (those for which $c(T) = T$) [$<$ should be $\leq$]

- Shows that modeling must be validated very carefully

- Mechanization can help: show that defined predicates can be both satisfied and falsified, and that axioms are satisfied by intended models (also establishes their consistency)

## Clock Synchronization (ctd. 2)

- Erwin Liu designed and formally verified a circuit to perform part of the algorithm

- Found the assumption that initial clock adjustments are all zero is unrealistic

- Deleted this assumption from the original verification and reran all proofs

- Found that proofs of some internal lemmas need adjusting, but main argument is unchanged

## Clock Synchronization (ctd. 3)

- There are many clock synchronization algorithms
- Shankar verified general treatment due to Fred Schneider that includes almost of them (using EHDM)

  Establishes correctness of all algorithms in a certain class satisfying 11 assumptions
- Found and corrected several small errors during verification
- And showed interactive convergence satisfies the assumptions
- Paul Miner argued that one of the assumptions is very difficult to establish of a given algorithm (almost as hard as proving synchronization)
- He showed that this assumption could be discharged once and for all from suitably modified versions of the other 10
- And did it for the Lundelius-Lynch algorithm

## Formal Verifications Performed: Distributed Consensus

Oral Messages algorithm for distributed consensus (aka. Byzantine Agreement—distribution of consistent sensor values in the presence of faults)

- First verified by Bevier and Young using Boyer-Moore prover (they also verified its optimality and an implementation)
- We found a rather simpler treatment and used it as a test case in developing PVS (it's an order of magnitude simpler than clock synchronization)
- Then studied important variation (developed at Allied Signal for MAFT flight control computer) that uses "Hybrid" fault model

## Hybrid Fault Models

- Classically, algorithms for state-machine replication consider components to be either faulty or nonfaulty
- No assumptions made about faulty components
- Tolerates any *kind* of fault, but requires a lot of redundancy, so cannot tolerate a large *number* of faults
  (for a given level of replication)
- "Hybrid" fault models include "arbitrary" fault mode, but also consider some common, simple kinds of fault ("manifest" and "symmetric") and can tolerate more of these
- "Hybrid" algorithms are strictly superior to pure "Byzantine" ones (e.g., $n > 3a + 2s + m$ rather than $n > 3f$)
- But their analysis is much more complex

## Distributed Consensus (ctd. 1)

- Algorithm Z, published with detailed proof of correctness by Thambidurai and Park, performs distributed consensus under hybrid fault model
- Lincoln and Rushby found a bug in the algorithm while attempting formal verification (implementation in MAFT is slightly different, and is OK)
- Developed modified algorithm and verified its correctness

  (Verified an *incorrect* variant on the way—bug was masked by an incorrect axiomatization of "hybrid majority vote")
- Subsequently adapted this algorithm to Draper FTP architecture and verified it in a couple of days

## Clock Synchronization (Again)

Returned to clock synchronization to extend benefits of hybrid fault model to that problem

- Original verification had taken a couple of months, has over 200 lemmas
- Extension to hybrid case requires several small adjustments (e.g., faulty/nonfaulty dichotomy must be revised)
- Consideration of additional cases in Lemma 5
- And a more complicated counting argument in the main theorem
- Overall, a couple of day's work
- Algorithm is about the best known; also does well against link faults

## Distributed Consensus (Again)

- Hybrid oral messages algorithm doesn't do well against link faults
- There is another class of distributed consensus algorithms that use authenticated communications: "lieutenants" cannot lie
- But they break if authentication is flawed
- Can also add authentication to Oral Message algorithms
- Simple worst-case analyses do not distinguish between the various candidate algorithms
- Cannot find good characterizations of behavior in presence of link faults

## Exploring Design Alternatives with Formal Methods

- Mur$\phi$ is an explicit state exploration tool developed by David Dill at Stanford
  - Given a specification of a finite-state concurrent system, explores *all* behaviors by brute-force
- Used Mur$\phi$ to do exhaustive "symbolic fault injection" for five-processor configuration
- Rediscovered the bug in Algorithm Z
- But found that ZA works very well
- Seems a generally useful way to evaluate fault-tolerant algorithms

## Formal Verifications Performed

Insufficient time to describe others:

- Verification of overall architecture and transient recovery
  - Done by Rick Butler, Ben DiVito and others at LaRC
  - The biggest verification done with EHDM (and probably the biggest by anyone using someone else's tools)
- Further properties of clock synchronization
  - Initial synchronization and transient recovery
  - Done by Paul Miner
- Circuit for clock synchronization
  - Verification used a combination of PVS, DDD, and a BDD-based tautology checker
  - Done by Paul Miner
- Diagnosis algorithms (done by Pat Lincoln in PVS)

## Lessons Learned (in *this* domain)

- None of what we have done is "program verification" (i.e., proving a program correct wrt. its detailed specifications)

- In this domain, concern centers on the *hardest and most difficult problems of design*: the redundancy management architecture and algorithms that ensure continued safe operation despite hardware failures

- Intrinsically difficult problems: reasoning about distributed, concurrent, real-time systems operating in the presence of faults
  - Number of possible behaviors is vast

## Lessons Learned (ctd. 1)

For intrinsically difficult problems and where systems can exhibit huge numbers of different behaviors

- Formal methods can add a lot of value: assurance through testing is totally infeasible, hand proofs are unreliable

- Need to understand quite a lot about the problem domain to apply formal methods usefully

- Need to be skilled at abstraction to apply formal methods effectively

- Can be accomplished by a few highly skilled people: don't need to train every programmer

## Lessons Learned (ctd. 2)

Mechanized formal methods provide much value in addition to proofs of correctness

- Reveal errors in published proofs and algorithms—i.e., discovery of *in*correctness

- Expose all assumptions and assist in eliminating them and in sharpening their statements

- Support development of streamlined arguments and enhanced understanding that can lead to improvements in algorithm or architecture

- Allow inexpensive and reliable exploration of alternative designs

- And adaptation to changed requirements or assumptions

## Lessons Learned (ctd. 3)

To achieve these benefits, mechanizations of formal methods must have certain attributes

- A civilized specification language

- Mechanisms for early error detection

- A *lot* of theorem-proving horsepower
  - That can be kept under control
  - And that provides more information than just "proved" and silence

- Mechanized deductive support in addition to theorem proving

- Support for changes and exploration

## The Design of PVS

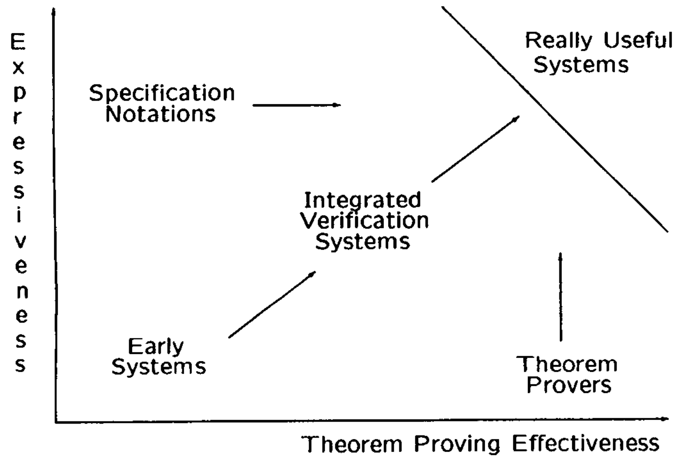
PVS had other influences besides these examples

- Hardware (Saxe pipeline, Tamarack, Cantu ALU, and AAMP5 examples)
- Requirements (Jet-Select example, A7 and SCR methods)
- Real-time (railroad crossing and other examples)
- Embeddings of other logics (Duration Calculus)
- Our previous systems (EHDM, Muse)
- Other systems (principally HOL, Boyer-Moore, and SMV)

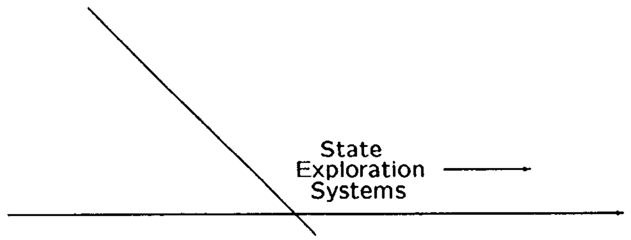
## A Civilized Specification Language

- A specification language provides some of the features and conveniences of a programming language (e.g., declarations, packaging in modules, parameterization)
- And computer-science data types (e.g., numbers, tuples, records, functions, sets, lists, trees)
- And should use a reasonably familiar syntax
- But it must also support deduction—i.e., it must be a logic
- We want that logic to be expressive, and we also want deduction to be automated very effectively
- Conflict: expressiveness and effectively automated deduction are usually inversely related
- Reconciling these disparate requirements is the fundamental challenge in building tools for formal methods

## The Three Traditions



## And A Fourth



## Reconciling the Conflicts: Foundations

- Traditional foundation for mathematics is first-order set theory
- And Hilbert-style presentation
- Designed by logicians for *studying* formal systems, not for actually *doing* formalization
  - o Requires metamathematical assistance (schema) to specify many important concepts (e.g., induction)
  - o And functions are inherently partial: inimical to efficient theorem proving
- For mechanized formal methods, we have different needs, and should therefore make different choices
  - o Frameworks (LF, Calculus of Constructions etc.)
  - o Higher-Order Logic (Simple Theory of Types)

## Reconciling the Conflicts: Pragmatics

- You cannot design a specification language and then hope to add effective mechanization
- And you cannot build a theorem prover and hope to persuade very many people that its logic is a specification language
- The formal method, its specification language, and its tools have to be designed together
  - o Tradeoffs and compromises are necessary
  - o But new opportunities arise, too
- Example: Enhancing the error-detection of typechecking

## Errors in Formal Specifications

- Most of the time spent with a formal methods tool goes in discovering and correcting errors
  - o Misunderstanding the requirements
  - o Flawed assumptions
  - o Faulty design
  - o Bugs in the formal specification
- Writing correct formal specifications is no easier than writing correct programs (for most people it's harder)
- Most formal specifications have errors; many are full of errors; a large number are meaningless (e.g., inconsistent)
- May be difficult to believe this, until you've experienced the personal shock of finding errors in your own specifications
- Tools should help you find errors early and cheaply

## Typechecking Formal Specifications

- Many errors in specifications can be found by strict typechecking—just as in programming languages
- But unlike programming languages, type systems of specification languages need not be restricted to those that are trivially decidable
- Can assume a theorem prover is available and allow typechecking to become undecidable
- This eliminates many of the criticisms of those who consider strict typing too restrictive
- Allows a lot of the specification to be expressed in the types, so that typechecking becomes a very strong check on consistency

## Example: Predicate Subtypes

- These are subtypes that are associated with a predicate
  Example (in PVS):

```
even_nat: TYPE = {i: nat | ∃ (j: nat): i = 2×j}
half: [even_nat → nat]
half_halves: AXIOM ∀ (e: even_nat): 2×half(e) = e
```

- Expressions of the supertype allowed where the subtype is expected, provided defining predicate can be discharged in that context. For example,

```
half_twice: LEMMA ∀ (i: nat): half(2×i) = i
```

- Generates the type-correctness condition (TCC)

```
TCC1: OBLIGATION ∀ (i: nat): ∃ (j:nat): 2×i = 2×j
```

## Example: Predicate Subtypes (Higher-Order)

- It often contributes clarity and precision to a specification if functions are identified as injections, surjections, etc.
- But how do we ensure that a purported surjection really has that property?
- Predicate subtypes! The surjections are a subtype of the functions associated with the following predicate

```
function_props[dom, rng: TYPE]: THEORY
  surjective?(f): bool = ∀ (r: rng): ∃ (d: dom): f(d) = r
```

So that

```
half: (surjective?[even_nat, nat]) = λ (e: even_nat): e/2
```

Generates the proof obligation

```
TCC2: OBLIGATION ∀ (n: nat): ∃ (e: even_nat): half(e) = n
```

## Consistency of Formal Specifications

- It is disturbingly easy to write inconsistent axioms
- So some specification languages forbid (or strongly discourage) them, in favor of constructive definitions that are guaranteed conservative
  - o But this is often too restrictive
    - ⋆ Eliminates property-oriented specifications
  - o And sometimes inappropriate
    - ⋆ Want to describe the assumed environment, not implement it
- Formal methods tools should enable the consistency of axiomatic specifications to be demonstrated (e.g., exhibit a model)
- But should also provide a rich collection of definitional forms

## Definitional Forms

- Can increase the variety and richness of definitional forms if prepared to use theorem proving to check them
- For example, recursive function definitions
  - o Without theorem proving, may be restricted to the syntactic form of primitive recursion
  - o With theorem proving, can accept definitions if arguments can be proved to decrease according to some well-founded ordering across recursive calls (note, system may need the sub-$\epsilon_0$ ordinals if lexicographic orderings are to be accepted)
- Other desirable forms
  - o Recursively defined abstract data types ("free types")
  - o Inductively defined functions
- These structured definitional forms also help theorem proving

## Example: Recursive Definition

- Here's the familiar factorial function

```
factorial(n: nat): RECURSIVE nat =
   IF n=0 THEN 1 ELSE n×factorial(n-1) ENDIF MEASURE n
```

  The termination TCC is what we expect:

```
TCC3: OBLIGATION ∀ (n: nat): n≠0 IMPLIES n-1 < n
```

- We also have a subtype TCC for `nat` from `n-1`:

```
TCC4: OBLIGATION ∀ (n: nat): n≠0 IMPLIES n-1 ≥ 0
```

  (The natural numbers are `nat: TYPE = { n: int | n ≥ 0 }`)

- Note how, together, these would identify the faulty termination condition `n=1`

## Design Choices in PVS Specification Language

- Specification language is a higher-order logic
  - With parameterized theories
  - Computer-science type-constructors (records, lists etc.)
    Also tables with checks on row/column well-formedness
  - Predicate subtypes and dependent types
  - Definitional forms include recursive functions, abstract data types, inductive definitions
- Built-in "prelude" and loadable "libraries" of domain-specific theories
- Emacs interface and extensive browsing and reporting capabilities
- LaTeX-prettyprinter

## Theorem Proving

- Tools for formal methods are built on theorem proving
- It is essential to make use of the insights and state-of-the-art techniques of that field
- Bit must recognize that the goals and assumptions of the theorem-proving community are not perfectly aligned with the needs of formal methods tools
  - Their goal is unassisted automation
  - They focus on proving true theorems
  - They automate raw logics, not specification languages

  Pure theorem provers are like dragsters; automating formal methods is more like Formula 1—many more challenges
  - But must go really fast when it's flat and straight
- Major choice: automatic or interactive theorem proving?

## Automatic Theorem Proving: The Downside

- There are no fully automatic theorem provers
- All require human guidance in indirect forms, such as
  - Weighting of literals, function symbols
  - Strategy selection
  - Orientation of equations
  - Invention and ordering of lemmas
  - Induction hints

  These have more to do with the prover than the proof
- User has to experiment to get them right
- Little feedback; little help discovering falsehoods
- And when proof is done, neither man nor machine is any wiser

## Automatic Theorem Proving: The Upside

There are a number of proven ideas and techniques that work remarkably well on specialized problem domains

- Propositional logic: BDDs (ordered binary decision diagrams)
- Pure equality: congruence closure
- Linear arithmetic: integer/linear programming (loop residue, SUP-INF)
- Datatypes, algebra: rewriting
- Modal/Temporal logics: model checking ($\mu$-calculus)
- Predicate calculus: resolution, unification
- Induction: Boyer-Moore heuristics
- Higher-Order: restricted forms of higher-order unification

Seldom encounter pure problems in practice: the engineering challenge is integrating these methods to solve mixed problems

## Interactive Proof Checking: The Downside

- Interactive guidance is an alternative to heroic automation
- Many interactive proof checkers are all interaction and no automation
  - o Enormous human investment to prove small theorems
  - o Infeasible to prove large or hard theorems
- Use many lemmas, large numbers of commands
  - o Hard for user to focus on the big picture during the proof
  - o Essence of the proof is lost in the details
  - o Proofs are not robust in the face of even small changes
- The users's concentration is the most precious resource: no excuse for wasting human time on trivial details if they can be blown away by automation

## Interactive Proof Checking: The Upside

- Interaction sustains the illusion of progress
- And puts the user in control
- Easier to learn and to master
  - o Provide there aren't too many different proof commands
- Can provide helpful feedback from failed proofs
- Can be customized with tactics (programs or macros that build larger proofs steps from the basic ones)

## LCF-Style Tactics

- Given a repertoire of primitive inference steps
- Construct more powerful steps by writing programs (tactics) that use the primitive ones
- Soundness depends only on the primitive inferences
- "Pure" interpretation (HOL)
  - o Basic steps should correspond to elementary rules of logic
  - o Tactics use a general programming language (ML), reduce to large numbers of primitive inferences
- "Impure" interpretation (PVS)
  - o Primitive steps should be chosen to provide the foundation for an efficient and powerful system
  - o Tactics are simple combinations of primitive ones

## The "Pure/Impure" Tradeoff With Tactics: 1

- Raw speed and power:
  - o The pure systems must synthesize decision procedures from primitive inferences: much slower in practice than the built-in procedures of impure systems
  - o And less powerful, because do not retain state, share data structures
- Effectiveness:
  - o The pure systems have large numbers of higher-level tactics, hard to remember what they do, tend to be either very specialized or not very powerful

## The "Pure/Impure" Tradeoff With Tactics: 2

- Soundness:
  - o Admittedly more challenging to gain confidence in the soundness of powerful inferences, but not exactly hard (most of what they do is search: affects completeness and termination, not soundness)
- Overall:
  - o Impure systems considerably more effective in practice
  - o Can always find better uses for machine cycles during interactive proof development than grinding out primitive inferences
  - o Use a "pure" second pass if paranoid

## The Theorem Proving Lifecycle During Formal Development

There is a lifecycle to theorem proving during formal development

Need different capabilities for each stage

- Exploration: must fail quickly, be able to stay in the loop
  Specialized methods are very useful
  - o Animation (direct execution/simulation)
  - o State exploration (reachability analysis)
  - o "Backwards execution" (fault-tree analysis)
  - o Efficient methods for specialized cases (model checking)
- Development: efficiency!
  Plus ability to comprehend overall structure
- Presentation: want a real proof
- Maintenance and Generalization: robustness

## Design Choices in PVS Prover

- Theorem prover is interactive
  - o Uses sequent calculus presentation
  - o Decision procedures for ground linear arithmetic and equality
  - o Hashing conditional rewriter integrated with these
  - o Strategy language for defining higher-level proof procedures
- Theorem prover can call external BDD package
  - o For simplification of large propositional structures
  - o And to decide formulas in $\mu$-calculus
    Used to automate CTL model checking (like SMV)
- Lemmas can be proved in any order
  (and introduced and modified on the fly)
- Proof dependency analysis keeps you honest
- Graphical displays of proof trees help comprehend big picture

## Conclusions

- Verifications described were once *tours de force*, now routine
- Provided benefits over and above verification: debugging, exploration, adaptation
- Startup costs of applying formal methods to a new application area can be quite high
  - Challenge is usually in the modeling and abstraction, not the mechanization
  - Requires skilled people
- Subsequent exploitation inexpensive, with very high added value
  - Exploration of design alternatives, adapting to changes
  - Reuse in subsequent designs
- Mechanized formal methods are ready for industrial exploitation in suitable circumstances

## To Learn More

- Browse papers and technical reports in `/pub/reports` on `ftp.csl.sri.com` (start with `readme.txt` and `abstracts.dvi.Z`), or use Mosaic or other WWW viewer on `http://www.csl.sri.com/sri-csl-fm.html`
  - CSL-95-1 is a chapter for the FAA Digital Systems Validation Handbook that discusses issues in formal methods and assurance
  - CSL-93-7 is a much longer and more technical treatment
- You can get PVS by anonymous FTP from `ftp.csl.sri.com` in `/pub/pvs` (also from WWW link at URL above)
  - Allegro Lisp for SunOS 4.1.3 (Solaris 2 available soon); Need 32M memory, 100M swap space, Sparc 2 or better
  - PVS2 is available for testing by current users; General release awaits completion of documentation

*3rd NASA Langley Formal Methods Workshop*

# Formal Methods Demonstration Project for Space Applications

May 11, 1995

John C. Kelly, Ph.D. and Ben DiVito, Ph.D.

*- Joint NASA Code Q RTOP -*

Jet Propulsion Laboratory*

Johnson Space Center

Langley Research Center

* The work described in this presentation was partially carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

## Contents

Overview of the Demonstration Project and the Guidebook - John C. Kelly

- The Software Requirements Problem
- Background Project Information
- Phases of the Project
- The NASA Formal Methods Specification and Verification Guidebook

Detailed look at applying Formal Methods to the Shuttle's GPS Upgrade Task - Ben DiVito

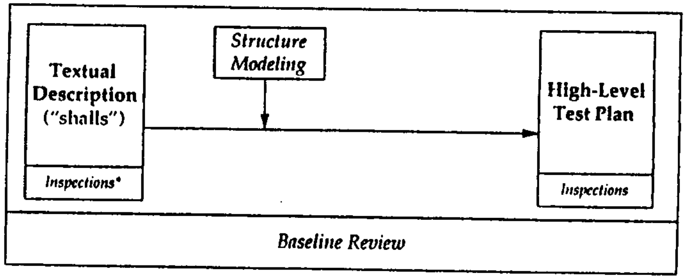
- Description of GPS CR Subset
- PVS Modeling of Principal Functions
- Receiver State Processing
- Results and Feedback
- Summary

## The Requirements Problem in Engineering Software Subsystems

Requirements and design specifications are a high priority candidate for better software engineering techniques

- Most hazardous software safety errors found during system integration and test of two NASA spacecraft were the result of requirements discrepancies or interface specifications [Lutz93].
- The highest density of major defects found through the use of software inspections was during the requirements phase. This was 7 times higher than the density of major defects found in code inspections [Kelly92].
- Requirements errors are between 10 and 100 times more costly to fix at later phases of the software lifecycle than at the requirements phase itself [Basili84], [Boehm84], [Kelly92].

## Current Requirements Process without Formal Methods



* Note: The word "inspection" is used differently in software than in manufacturing or hardware context. In software the term "inspection" refers to a structured peer review that uses a checklist and includes the collection of defect statistics. In software development, inspections are an upstream quality enhancement technique which support the principles of TQM (refer to NASA Standard 2202-93 or the two JPL Professional Development courses on this topic).

## History of Formal Methods POP

| | |
|---|---|
| Spring 1992 | - JSC, JPL and LaRC submit independent Formal Methods proposals to NASA Code Q |
| Summer and Fall '92 | - Code Q funds a Feasibility Study to derive a plan for how the 3 Centers could jointly do a Formal Methods RTOP |
| Winter '93 | - Formal Methods Demonstration Project officially starts with a kick-off meeting at JSC |
| October '93 | - Deliverables for Phase I completed (Formal Methods Specifications, Proofs, etc.) |
| December '93 | - Phase I Report completed for demonstration of Formal Methods on an existing spacecraft |
| FY94 | - Phase II Case Studies<br>- Formal Methods Guidebook (Vol 1) |
| FY95 | - Phase III Case Studies<br>- Formal Methods Guidebook (Vol 2) |

## Purpose

- The Goal of this study is to demonstrate the applicability of Formal Methods techniques on critical NASA software subsystems
- Phase I Task: Demonstrate the Applicability of Formal Methods to Shuttle's On-Board Jet Select Software Subsystem (A highly critical, yet relatively stable set of requirements)
- Phase II Tasks: Demonstrate Formal Methods on several smaller projects which are currently developing critical software and provide guidance at the managerial level
- Phase III Tasks: Demonstrate Formal Methods on a large critical project in the early development stages, demonstrate at the design level, and provide guidance at the technical level

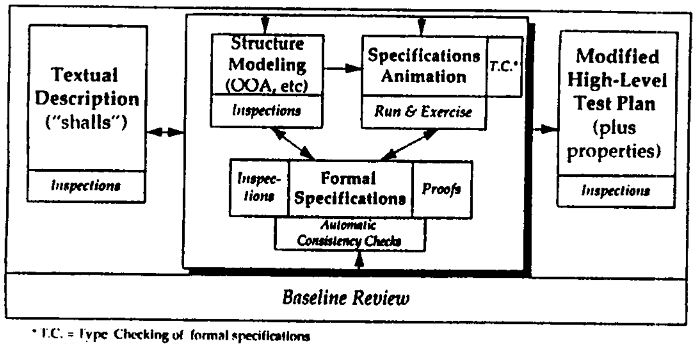
## Introduction: Team Members

- Jet Propulsion Laboratory
  - John Kelly, Ph.D., Rick Covington, Ph.D., Robyn Lutz, Ph.D., Ken Abernethy , Ph.D. (Furman U.)
- Johnson Space Center
  - Ernie Fridge, Dan Bowman (LORAL), Mike Beims (LORAL-Shuttle RA), Chris Hickey (LORAL-Shuttle RA)
- Langley Research Center
  - Ben DiVito, Ph.D. (VIGYAN), Judith Crow, Ph.D. (SRI), Rick Butler
- NASA HQ Sponsor: Alice Robinson
- Alumni
  - Betty Cheng, Ph.D. (MSU), Mori Khorrami (JPL), Doc Shankar, Ph.D. (IBM), Scott French (LORAL), Sally Johnson (LaRC), John Rushby, Ph.D. (SRI), Sam Owre (SRI), Al Nikora(JPL), Brent Auernheimer, Ph.D. (CSUF), Yoko Ampo (NEC), David Hamilton (HP)

## Generic approach for using Formal Methods to analyze requirements for Space Applications

- Select a portion of the requirements which is mission critical
- Model the selected requirement subset's structure using an informal approach (Object Modeling Techniques was used on some of the pilots)
- Develop Formal Specification for required functionality and state behavior (PVS was used for this on most of the pilots)
- Translate desired properties of the subsystem into a formal language then attempt to verify them against the Formal Specification with a theorem prover (PVS was used in most cases)
- Feedback lists of issues to Requirements Analysts and Development Engineers throughout this process

## Integrated Formal Methods Process


Textual Description ("shalls")
Inspections
Structure Modeling (OOA, etc)
Inspections
Specifications Animation
T.C.*
Run & Exercise
Inspec-tions
Formal Specifications
Proofs
Automatic Consistency Checks
Modified High-Level Test Plan (plus properties)
Inspections
Baseline Review

* T.C. = Type Checking of formal specifications
** OOA = Object Oriented Analysis

## Project Tasks and Deliverables

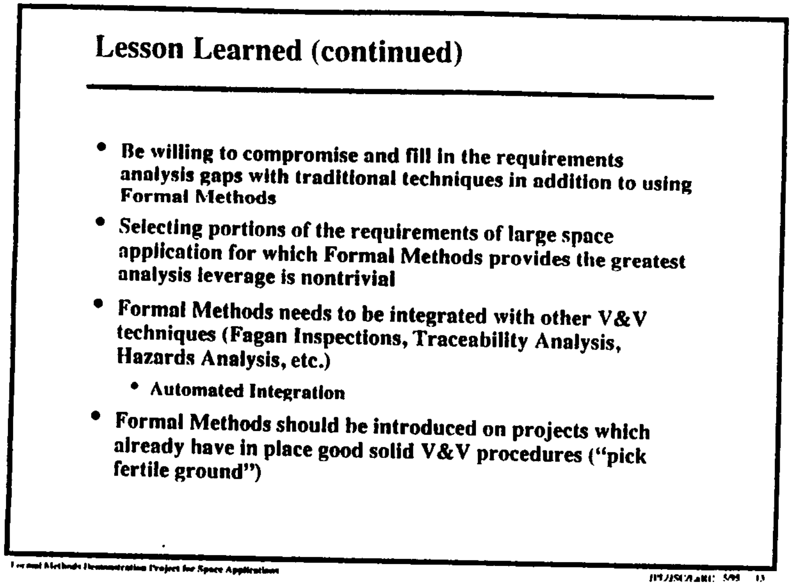
- Pre-project
  - Formal Methods Feasibility Study - October 1992
- Phase I: FY 1993
  - Shuttle's Jet Select Subsystem
    - 3 Levels of Formal Specifications
    - Ada emulator
  - Formal Methods Case Study Report
- Phase II: FY 1994
  - Shuttle's
    - MIR Docking Change Request
    - Orbit Digital Auto Pilot Model
    - Three Engine Out CR
    - Global Positioning System Change Request (ongoing)

## Project Tasks and Deliverables (continued)

- Phase II: FY 1994 (continued)
  - Cassini Space Craft- Fault Protection (Requirements)
  - International Space Station Alpha FDIR (ongoing)
  - Formal Methods Guidebook - (development of Volume I)
- Phase III: FY 1995 (current)
  - Cassini Space Craft- Fault Protection (Design)
  - International Space Station Alpha FDIR
  - AP101S Assembler for Shuttle
  - Global Positioning System Change Request for Shuttle
  - Formal Methods Guidebook
    - Review Vol. I
    - Develop and Review Vol. 2

## Overview of Results and Lessons Learned

- Formal Methods uncovered significant issues in mature requirements
- Even in an unstable requirements environment, formal specifications were found to be beneficial
- Requirements Analysts and Developers were generally impressed by the thoroughness and insight of the *issues lists* produced by the pilot study team
- Object Modeling Technique provides a good road map before developing Formal Specification
- PVS is robust enough to be used for the practical application of Formal Methods to requirements

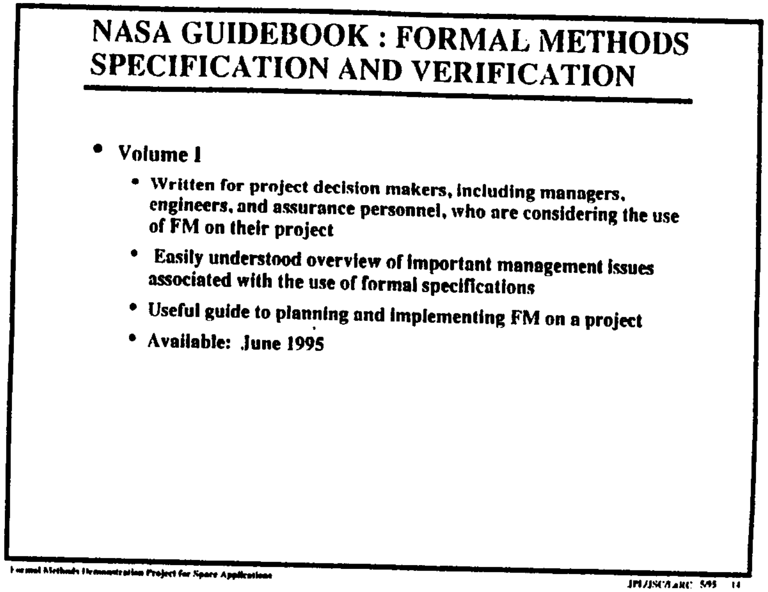


## Lesson Learned (continued)

- Be willing to compromise and fill in the requirements analysis gaps with traditional techniques in addition to using Formal Methods
- Selecting portions of the requirements of large space application for which Formal Methods provides the greatest analysis leverage is nontrivial
- Formal Methods needs to be integrated with other V&V techniques (Fagan Inspections, Traceability Analysis, Hazards Analysis, etc.)
  - Automated Integration
- Formal Methods should be introduced on projects which already have in place good solid V&V procedures ("pick fertile ground")

Formal Methods Demonstration Project for Space Applications

JPL/JSC/LaRC 5/95 13



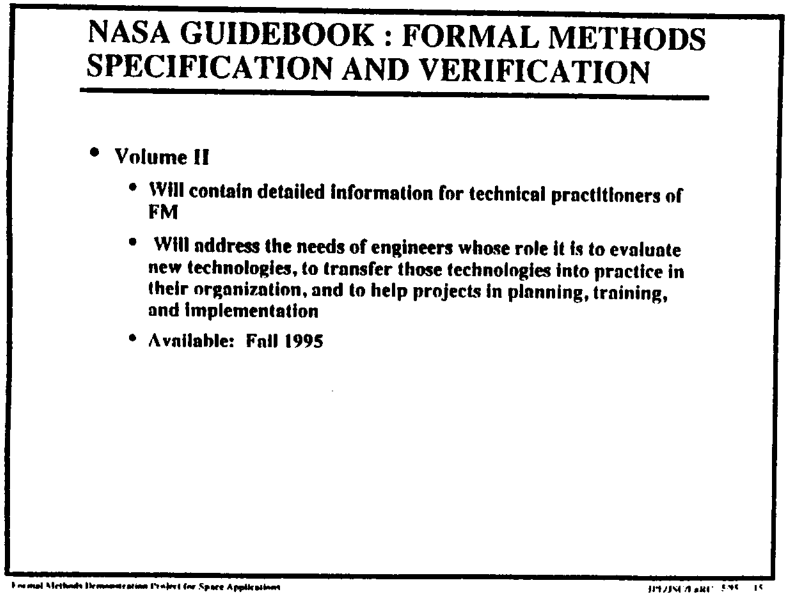
## NASA GUIDEBOOK : FORMAL METHODS SPECIFICATION AND VERIFICATION

- Volume I
  - Written for project decision makers, including managers, engineers, and assurance personnel, who are considering the use of FM on their project
  - Easily understood overview of important management issues associated with the use of formal specifications
  - Useful guide to planning and implementing FM on a project
  - Available: June 1995

Formal Methods Demonstration Project for Space Applications

JPL/JSC/LaRC 5/95 14

C-2.

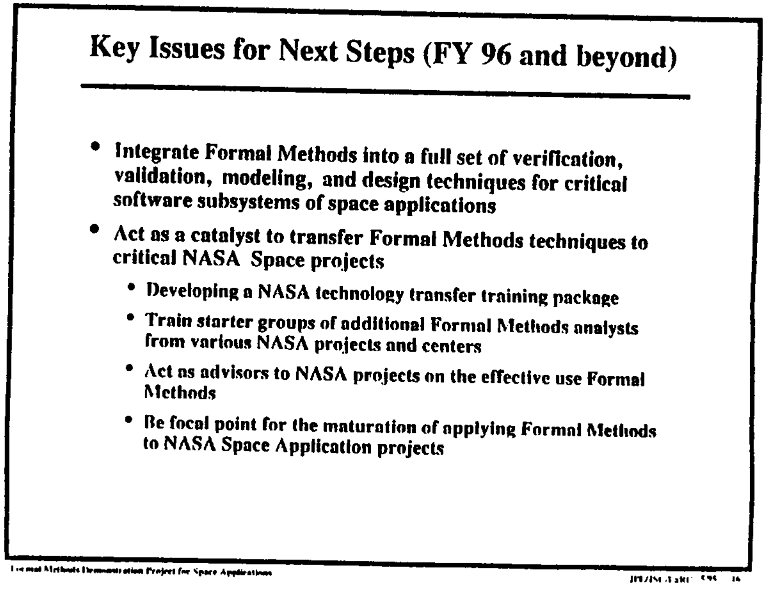


## NASA GUIDEBOOK : FORMAL METHODS SPECIFICATION AND VERIFICATION

- Volume II
  - Will contain detailed information for technical practitioners of FM
  - Will address the needs of engineers whose role it is to evaluate new technologies, to transfer those technologies into practice in their organization, and to help projects in planning, training, and implementation
  - Available: Fall 1995

Formal Methods Demonstration Project for Space Applications

JPL/JSC/LaRC 5/95 15



## Key Issues for Next Steps (FY 96 and beyond)

- Integrate Formal Methods into a full set of verification, validation, modeling, and design techniques for critical software subsystems of space applications
- Act as a catalyst to transfer Formal Methods techniques to critical NASA Space projects
  - Developing a NASA technology transfer training package
  - Train starter groups of additional Formal Methods analysts from various NASA projects and centers
  - Act as advisors to NASA projects on the effective use Formal Methods
  - Be focal point for the maturation of applying Formal Methods to NASA Space Application projects

Formal Methods Demonstration Project for Space Applications

JPL/JSC/LaRC 5/95 16

S2-61

N96-10028

2

# FORMAL METHODS DEMONSTRATION PROJECT FOR SPACE APPLICATIONS

***Ben L. DiVito***

P. 7

VÍGYAN, Inc

The Space Shuttle program is cooperating in a pilot project to apply formal methods to live requirements analysis activities. As one of the larger ongoing Shuttle Change Requests (CRs), the Global Positioning System (GPS) CR involves a significant upgrade to the Shuttle's navigation capability. Shuttles are to be outfitted with GPS receivers and the primary avionics software will be enhanced to accept GPS-provided positions and integrate them into navigation calculations. Prior to implementing the CR, requirements analysts at Loral Space Information Systems, the Shuttle software contractor, must scrutinize the CR to identify and resolve any requirements issues.

We describe an ongoing task of the Formal Methods Demonstration Project for Space Applications whose goal is to find an effective way to use formal methods in the GPS CR requirements analysis phase. This phase is currently under way and a small team from NASA Langley, ViGYAN Inc. and Loral is now engaged in this task. Background on the GPS CR is provided and an overview of the hardware/software architecture is presented. We outline the approach being taken to formalize the requirements, only a subset of which is being attempted. The approach features the use of the PVS specification language to model "principal functions," which are major units of Shuttle software. Conventional state machine techniques form the basis of our approach.

Given this background, we present interim results based on a snapshot of work in progress. Samples of requirements specifications rendered in PVS are offered for illustration. We walk through a specification sketch for the principal function known as GPS Receiver State Processing. Results to date are summarized and feedback from Loral requirements analysts is highlighted. Preliminary data is shown comparing issues detected by the formal methods team versus those detected using existing requirements analysis methods. We conclude by discussing our plan to complete the remaining activities of this task.

## Using Formal Methods to Analyze the Space Shuttle's GPS Change Request

Ben L. Di Vito

ViGYAN, Inc.
30 Research Drive
Hampton, VA 23666

3rd NASA Langley Formal Methods Workshop
11 May 1995

## Shuttle Program Background

- Contractor organization
  - Rockwell International – Shuttle prime contractor
  - Loral Space Info. Sys. (was IBM) – Software contractor
  - Draper Lab – Experts in Guidance, Navigation and Control
- Software modifications are packaged as Change Requests (CRs)
  - Usually modest in scope, localized in function
  - Provide capabilities to meet specific mission needs
- Software releases are called Operational Increments (OIs)
  - Include one or more CRs – issued around once per year
- Requirements Analysis
  - Conducted by Loral Requirements Analysts (RAs) prior to turning CR over to development team
  - Once in development, problems are more costly to fix

## Global Positioning System (GPS)

GPS is a satellite-based navigation system operated by DoD

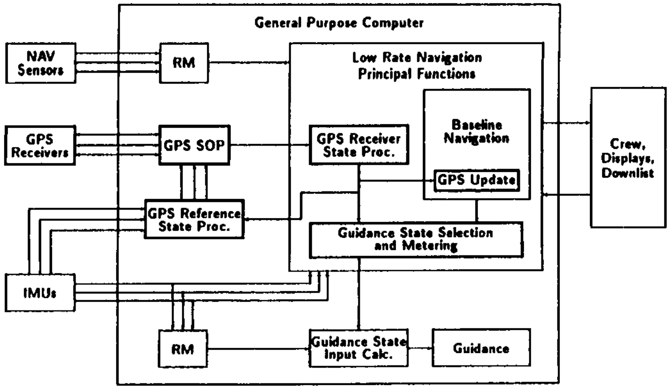
- Constellation of 24 satellites in high orbits
- Receive-only system requires dedicated hardware
  - Need to track 4 or more satellites simultaneously
  - Separate signals need to be recovered after undergoing code division multiplexing
- Receivers solve for position and velocity (and time)
  - Standard Positioning Service gives 100m accuracy
  - Precise Positioning Service gives 10m accuracy
- DoD phasing out TACAN navigation system by 2000

## GPS Change Request

Shuttle Program is undertaking a large and complex CR to add GPS navigation capabilities to the Shuttle fleet

- Motivation: loss of TACAN navigation system
  - Need equivalent navigation aid during entry and landing
- Two-phase integration plan
  - Single-string implementation (one receiver)
  - Full-up implementation (three receivers)
- Integrated architecture
  - GPS receivers provide navigation data to General Purpose Computers (GPCs)
  - Several new Principal Functions added to software
  - Many smaller changes made to existing navigation software
- GPS CR more complex than typical CR

## GPS Integrated Architecture



## Approach to Applying Formal Methods

We are pursuing a selective application of formal methods

- Focus on core subset involving new principal functions
  - GPS Receiver State Processing
  - GPS Reference State Processing
- Use PVS to model principal function requirements
  - Derive stylized specification approach tailored to Shuttle software
  - Maintain traceability to existing requirements
  - Emphasize readability by nonexperts
- Formulate properties and attempt proving later, if warranted
  - Possible candidate is feedback loop from receivers to Receiver State Processing and back

## Description of GPS CR Subset

Principal functions are decomposed into subfunctions

- Receiver State Processing
  - GPS IMU Assign
  - GPS Navigation State Propagation
  - GPS State Vector Quality Assessment
  - GPS State Vector Selection
  - GPS Reference State Announced Reset
- Reference State Processing
  - GPS External Data Snap
  - IMU GPS Selection
  - GPS Reference State Initialization and Reset
  - GPS Reference State Propagation

## PVS Modeling of Principal Functions

Principal functions are regularly scheduled software entities

- Execution environment provides a large variable space that is read from and written into
- Interface is defined by explicit inputs and outputs enumerated in tables
- Principal functions are composed of several subfunctions invoked sequentially
  - Inputs are "passed down" to subfunctions
  - Outputs are "passed up" from subfunctions
- Local variables may be either transient or persistent
- We use a simple state machine model encoded in PVS
  - Principal function is represented as a single state-transition function

$$M : I \times S \rightarrow [O \times S]$$

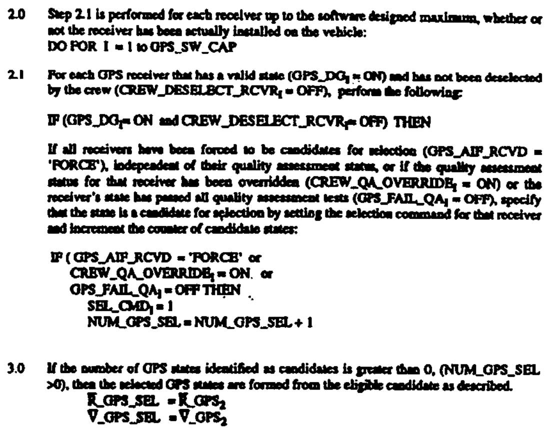
## PVS Modeling of Principal Functions (Cont'd)

**Abstract structure in PVS notation:**

```
pf_result: TYPE = [# output: pf_outputs, state: pf_state #]

principal_function (pf_inputs, pf_state,
                    pf_I_loads, pf_K_loads,
                    pf_constants) : pf_result =

      (# output := <output expression>,
         state  := <next-state expression>
      #)
```

- **Principal functions use two kinds of variable data (input values, previous-state values) and three kinds of constant data (I-loads, K-loads, constants)**
- **Executing a principal function produces output values and next-state values**
- **All side effects are to be captured by this model**

## Requirements for Receiver State Processing



2.0 Step 2.1 is performed for each receiver up to the software designed maximum, whether or not the receiver has been actually installed on the vehicle:
DO FOR I = 1 to GPS_SW_CAP

2.1 For each GPS receiver that has a valid state (GPS_DG$_I$ = ON) and has not been deselected by the crew (CREW_DESELECT_RCVR$_I$ = OFF), perform the following:

IF (GPS_DG$_I$= ON and CREW_DESELECT_RCVR$_I$= OFF) THEN

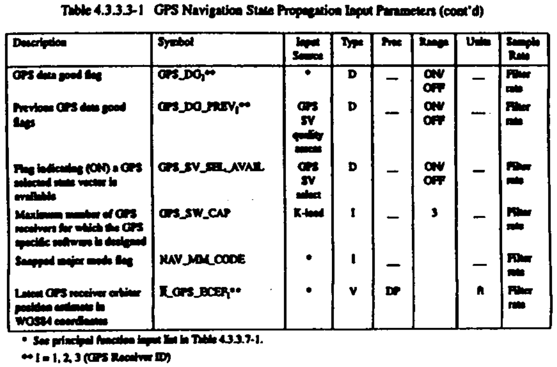
If all receivers have been forced to be candidates for selection (GPS_AIF_RCVD = 'FORCE'), indepeadent of their quality assessment status, or if the quality assessment status for that receiver has been overridden (CREW_QA_OVERRIDE$_I$ = ON) or the receiver's state has passed all quality assessment tests (GPS_FAIL_QA$_I$ = OFF), specify that the state is a candidate for selection by setting the selection command for that receiver and increment the counter of candidate states:

```
IF ( GPS_AIF_RCVD = 'FORCE' or
   CREW_QA_OVERRIDE_I = ON. or
   GPS_FAIL_QA_I = OFF THEN
      SEL_CMD_I = 1
      NUM_GPS_SEL = NUM_GPS_SEL + 1
```

3.0 If the number of GPS states identified as candidates is greater than 0, (NUM_GPS_SEL >0), then the selected GPS states are formed from the eligible candidate as described.
$\bar{R}$_GPS_SEL = $\bar{R}$_GPS$_2$
$\bar{V}$_GPS_SEL = $\bar{V}$_GPS$_2$

## Requirements for Receiver State Processing (Cont'd)

Table 4.3.3.3-1 GPS Navigation State Propagation Input Parameters (cont'd)



| Description | Symbol | Input Source | Type | Prec | Range | Units | Sample Rate |
|---|---|---|---|---|---|---|---|
| GPS data good flag | GPS_DG$_I$** | * | D | — | ON/ OFF | — | Filter rate |
| Previous GPS data good flags | GPS_DG_PREV$_I$** | GPS SV quality assess | D | — | ON/ OFF | — | Filter rate |
| Flag indicating (ON) a GPS selected state vector is available | GPS_SV_SEL_AVAIL | GPS SV select | D | — | ON/ OFF | — | Filter rate |
| Maximum number of GPS receivers for which the GPS specific software is designed | GPS_SW_CAP | K-load | I | — | 3 | — | Filter rate |
| Snapped major mode flag | NAV_MM_CODE | * | I | — |  | — | Filter rate |
| Latest GPS receiver orbiter position estimate in WGS84 coordinates | $\bar{R}$_GPS_ECEF$_I$** | * | V | DP |  | ft | Filter rate |

* See principal function input list in Table 4.3.3.7-1.

** I = 1, 2, 3 (GPS Receiver ID)

## Modeling of Receiver State Processing

**Selected data types rendered in PVS**

```
major_mode_code:     TYPE = nat
mission_time:        TYPE = real
GPS_id:              TYPE = {n: nat | 1 <= n & n <= 3}

receiver_mode:       TYPE = {init, test, nav}
AIF_flag:            TYPE = {auto, inhibit, force}

M50_axis:            TYPE = {Xm, Ym, Zm}
position_vector:     TYPE = [M50_axis -> real]
velocity_vector:     TYPE = [M50_axis -> real]
GPS_positions:       TYPE = [GPS_id -> position_vector]
GPS_velocities:      TYPE = [GPS_id -> velocity_vector]

GPS_predicate:       TYPE = [GPS_id -> bool]
GPS_times:           TYPE = [GPS_id -> mission_time]
GPS_FOM_vector:      TYPE = [GPS_id -> GPS_figure_of_merit]
tolerance_vector:    TYPE = [GPS_id -> real]
WGS84_to_EF_matrix:  TYPE = [earth_fixed_axis -> [WGS84_axis -> real]]
```

## Sample Subfunction of Receiver State Processing

```
ref_state_anncd_reset_out: TYPE = [#
      GPS_anncd_reset_avail:    GPS_predicate,
      GPS_anncd_reset:          GPS_predicate,
      R_ref_anncd_reset:        GPS_positions,
      T_anncd_reset:            mission_time,
      T_ref_anncd_reset:        mission_time,
      V_IMU_ref_anncd_reset:    velocity_vector,
      V_ref_anncd_reset:        GPS_velocities
      #]

ref_state_announced_reset(DT_anncd_reset,
                          GPS_DG,
                          GPS_SW_cap,
                          R_GPS,
                          T_anncd_reset,
                          T_current_filt,
                          T_GPS,
                          V_current_GPS,
                          V_GPS) : ref_state_anncd_reset_out
```

13

## Principal Function Interface Types

```
rec_sp_inputs: TYPE = [#
      crew_deselect_rcvr:       GPS_predicate,
       . . .                     . . .
      V_GPS_ECEF:               GPS_velocities_WGS84,
      V_IMU2_save:              GPS_velocities
      #]

rec_sp_state: TYPE = [#
      GPS_DG_prev:              .GPS_predicate,
       . . .                     . . .
      V_last_GPS_sel:           velocity_vector
      #]

rec_sp_I_loads: TYPE = [#
      acc_prop_min:             real,
       . . .                     . . .
      M_WGS84_to_EF:            WGS84_to_EF_matrix,
      sig_diag_GPS_nom:         cov_diagonal_vector
      #]
```

14

## Principal Function Interface Types (Cont'd)

```
rec_sp_K_loads: TYPE = [#
      GPS_SW_cap:              num_GPS
      #]

rec_sp_constants: TYPE = [#
      deg_to_rad:              real,
      earth_rate:              real,
      G0:                      real,
      nautmi_per_ft:           real
      #]

rec_sp_outputs: TYPE = [#
      corr_coeff_GPS:          real,
      DELR_ratio_QA2_display: GPS_ratios,
       . . .                    . . .
      V_ref_anncd_reset:       GPS_velocities
      #]

rec_sp_result: TYPE = [# output: rec_sp_outputs, state: rec_sp_state #]
```

15

## Principal Function Specification

```
GPS_receiver_state_processing((rec_sp_inputs:    rec_sp_inputs),
                              (rec_sp_state:     rec_sp_state),
                              (rec_sp_I_loads:   rec_sp_I_loads),
                              (rec_sp_K_loads:   rec_sp_K_loads),
                              (rec_sp_constants: rec_sp_constants) )
                              : rec_sp_result =
  LET
     IMU_assign_out =
        IMU_assign(
           V_curfent_filt         (rec_sp_inputs),
           V_IMU2_save            (rec_sp_inputs)),

     nav_state_prop_out =
        nav_state_propagation(
           acc_prop_min           (rec_sp_I_loads),
            . . .                  . . .
           V_GPS_ECEF             (rec_sp_inputs),
           V_last_GPS             (IMU_assign_out),
           V_last_GPS_sel         (rec_sp_state) ),
```

16

## Principal Function Specification (Cont'd)

```
SV_qual_assess_out =
    state_vector_quality_assessment(
        GPS_DG              (rec_sp_inputs),
         . . .               . . .
        V_GPS               (nav_state_prop_out) ),

state_vect_sel_out =
    state_vector_selection(
        corr_coeff_GPS_nom  (rec_sp_I_loads),
         . . .               . . .
        V_GPS_sel           (nav_state_prop_out) ),

ref_st_ann_reset_out =
    ref_state_announced_reset(
        DT_anncd_reset      (rec_sp_I_loads),
         . . .               . . .
        V_GPS               (nav_state_prop_out) )
```

## Principal Function Specification (Cont'd)

```
IN
  (# output := (#
       corr_coeff_GPS :=      corr_coeff_GPS      (state_vect_sel_out),
        . . .                  . . .               . . .
       GPS_fail_QA4 :=        GPS_fail_QA4        (SV_qual_assess_out),
       GPS_lat :=             GPS_lat             (state_vect_sel_out),
       GPS_lon :=             GPS_lon             (state_vect_sel_out),
        . . .                  . . .               . . .
       V_ref_anncd_reset :=   V_ref_anncd_reset   (ref_st_ann_reset_out)
       #),

    state := (#
       GPS_DG_prev :=         GPS_DG_prev         (SV_qual_assess_out),
       GPS_SV_sel_avail :=    GPS_SV_sel_avail    (state_vect_sel_out),
        . . .                  . . .               . . .
       V_GPS_sel :=           V_GPS_sel           (state_vect_sel_out),
       V_last_GPS_sel :=      V_last_GPS_sel      (nav_state_prop_out)
       #)
  #)
```

## Results (In Progress)

Experience with effort so far:

- Outlook is promising, but it's still early
- CR requirements are still converging
  - Another revision cycle is likely
  - After next cycle, FM results should be more meaningful
- FM-based review is helping requirements analysis
  - Many interface errors being detected
- PVS can be used effectively to formalize this application
  - Custom specification approach should be easy to duplicate
  - Good prospects for continuation by nonexperts
  - Specification activity assisted by tools, but doing manual specification is also feasible here

## Feedback from Requirements Analysts

Some RAs are optimistic about potential impact of FM

- Approach used is helpful in detecting two classes of errors:
  - "Requirements meet the CR author's intent; CR will work"
  - "Interfaces Documented and Consistent"
- Preliminary comparison with conventional process
  - Errors detected in Reference State Processing versus those also found by current process

| Error Severity | With FM | Existing |
|---|---|---|
| High Major | 2 | 0 |
| Low Major | 5 | 1 |
| High Minor | 17 | 3 |
| Low Minor | 6 | 0 |
| Totals | 30 | 4 |

## Continuation Plans

Plan for near term CR analysis

- Continue elaborating formal specifications using current style
  - Complete two principal functions under way
  - Consider adding other functions as resources allow
  - Collect feedback and data from RAs
- Go down to moderate level of detail
  - Incorporate enough detail to capture branching conditions
- Evaluate tradeoffs of formalizing subsystem properties
  - Properties based on sequences of state vectors
  - Limited proving may be worthwhile

## Summary

- Formal methods aiding requirements analysis on a significant Shuttle CR
- Specification techniques customized for Shuttle software
- So far, keeping up with requirements analysis process
  - Enables meaningful comparisons
- Early results are encouraging
- Remains to be seen whether techniques can be transferred to and adopted by RAs

omit

# Session 5: Software Systems (2)

**C. Michael Holloway, Chair**

- **Ada 9X Language Precision Team**, by *David Guaspari*, Odyssey Research Associates
- **Introduction to Penelope and Its Applications**, by *David Guaspari*, Odyssey Research Associates

# Formal Methods in the Design of Ada 95

*David Guaspari*

*Odyssey Research Associates*

Formal, mathematical methods are most useful when applied early in the design and implementation of a software system---that, at least, is the familiar refrain. I will report on a modest effort to apply formal methods at the earliest possible stage, namely, in the design of the Ada 95 programming language itself. This talk is an ``experience report" that provides brief case studies illustrating the kinds of problems we worked on, how we approached them, and the extent (if any) to which the results proved useful. It also derives some lessons and suggestions for those undertaking future projects of this kind

Ada 95 is the first revision of the standard for the Ada programming language. The revision began in 1988, when the Ada Joint Programming Office first asked the Ada Board to recommend a plan for revising the Ada standard. The first step in the revision was to solicit criticisms of Ada 83. A set of requirements for the new language standard, based on those criticisms, was published in 1990. A small design team, the Mapping Revision Team (MRT), became exclusively responsible for revising the language standard to satisfy those requirements. The MRT, from Intermetrics, is led by S. Tucker Taft.

The work of the MRT was regularly subject to independent review and criticism by a committee of Distinguished Reviewers and by several advisory teams---for example, by two User/Implementor teams, each consisting of an industrial user (attempting to make significant use of the new language on a realistic application) and a compiler vendor (undertaking, experimentally, to modify its current implementation in order to provide the necessary new features). One novel decision established the Language Precision Team (LPT), which investigated language proposals from a mathematical point of view.

The LPT applied formal mathematical analysis to help improve the design of Ada 95 (e.g., by clarifying the language proposals) and to help promote its acceptance (e.g., by identifying a verifiable subset that would meet the needs of safety-critical applications). The first LPT project, which ran from the fall of 1990 until the end of 1992, produced studies of several language issues: optimization, sharing and storage, tasking and protected records, overload resolution, the floating point model, distribution, program errors, and object-oriented programming. The second LPT project, in 1994, formally modeled the dynamic semantics of a large part of the (almost) final language definition, looking especially for interactions between language features.

# Formal methods in the design of Ada95

**David Guaspari**
**NASA Formal Methods Workshop**
**May 10-12, 1995**

**Odyssey Research Associates**
**301 Dates Drive**
**Ithaca, NY 14850-1326**
**(607) 277-2020**
**davidg@oracorp.com**

## Revising Ada

- ▢ Requirements solicited, published in 1990
- ▢ Mapping Revision Team (MRT)
  - ○ Small design team (as with Ada83)
  - ○ Led by S. Tucker Taft (Intermetrics)
- ▢ Advisory groups
  - ○ Distinguished Reviewers
  - ○ User/Implementor Teams
  - ○ Language Precision Team (1991-1992, 1994)

## Revisions Include

- ▢ Fixing infelicities
- ▢ Real-time programming (protected records)
- ▢ Object oriented programming (single inheritance)
- ▢ Hierarchical library

Of concern for reliable computing

- ▢ More predictable semantics in the face of errors
- ▢ Safety and security annex
- ▢ Consideration of formal methods in the design
- ▢ Establishment of Rapporteur Group for critical software

## Goals of the Language Precision Project

- ▢ Improve design and promote acceptance of Ada9X
  - ○ Clarify problems
  - ○ State and/or prove properties of the design
  - ○ Identify verifiable subset
  - ○ Provide basis for future formal work

## Technical reports

- ❐ Phase 1: language issues
  - ○ optimization
  - ○ sharing and storage
  - ○ protected records
  - ○ overload resolution
  - ○ the floating point model
  - ○ distribution
  - ○ program errors
  - ○ object-oriented programming
- ❐ Phase 2: descriptive
  - ○ "natural semantics" model
  - ○ sequential dynamic semantics

## This talk is an experience report

- ❐ Previous formal work on Ada
- ❐ Strategies
- ❐ Phase 1 case studies
- ❐ Phase 2 summary
- ❐ Conclusions

## Previous formal work on Ada

- ❐ Formal definition projects
  - ○ INRIA (official, little effect)
  - ○ Dansk Datamatik (led to DDC compiler)
  - ○ NYU (led to first validated compiler)
  - ○ DDC/CRAI (almost whole language)
- ❐ Formal definitions of subsets, reasoning systems
  - ○ Penelope (ORA)
  - ○ AVA (CLInc)
  - ○ Aerospace Corporation
  - ○ SPARK (Program Validation Ltd.)
  - ○ ProSpecTra
  - ○ RAISE (Computer Resources International)

## Phase 1: targets of opportunity

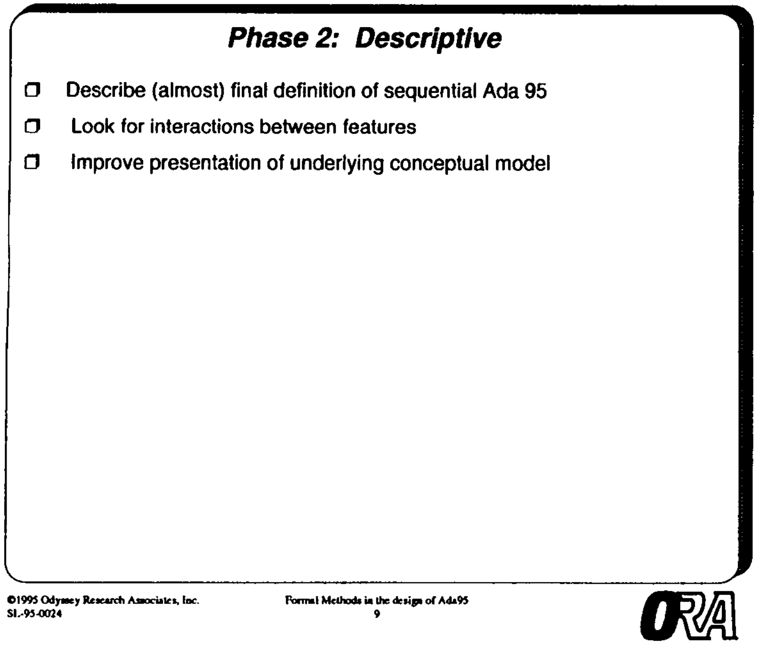
- ❐ Ignore well-understand parts of language
- ❐ Formalism and approach chosen "per problem"

Advantages

- ❐ Right level of abstraction
- ❐ Tractable: some hope of prompt response

Disadvantages

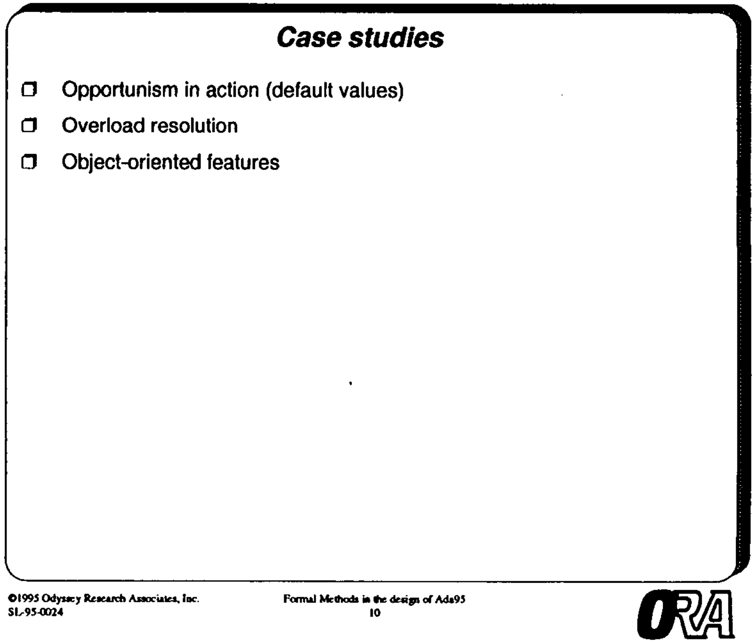
- ❐ Axiomatic method makes assumptions – how justified?
- ❐ Disregards interactions of features



## Phase 2: Descriptive

- □ Describe (almost) final definition of sequential Ada 95
- □ Look for interactions between features
- □ Improve presentation of underlying conceptual model

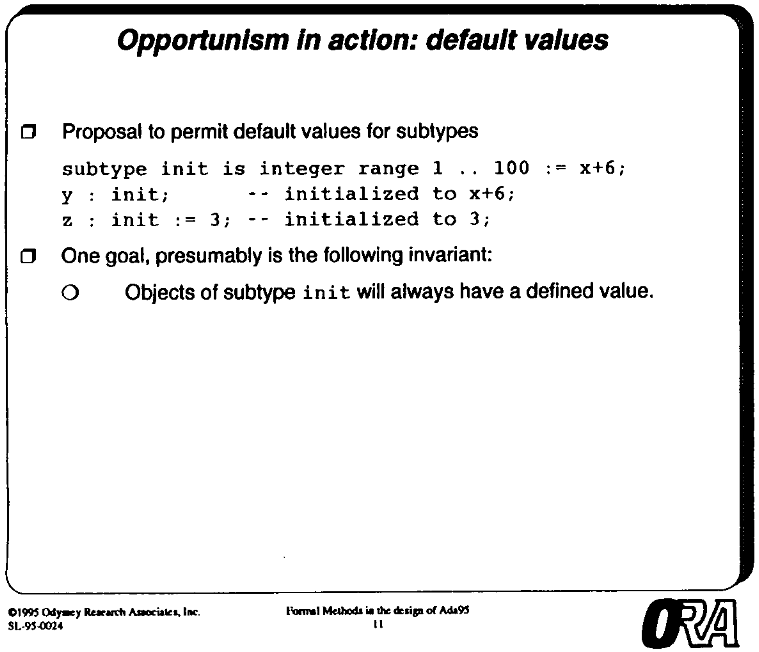
SL-95-0024



## Case studies

- □ Opportunism in action (default values)
- □ Overload resolution
- □ Object-oriented features

SL-95-0024



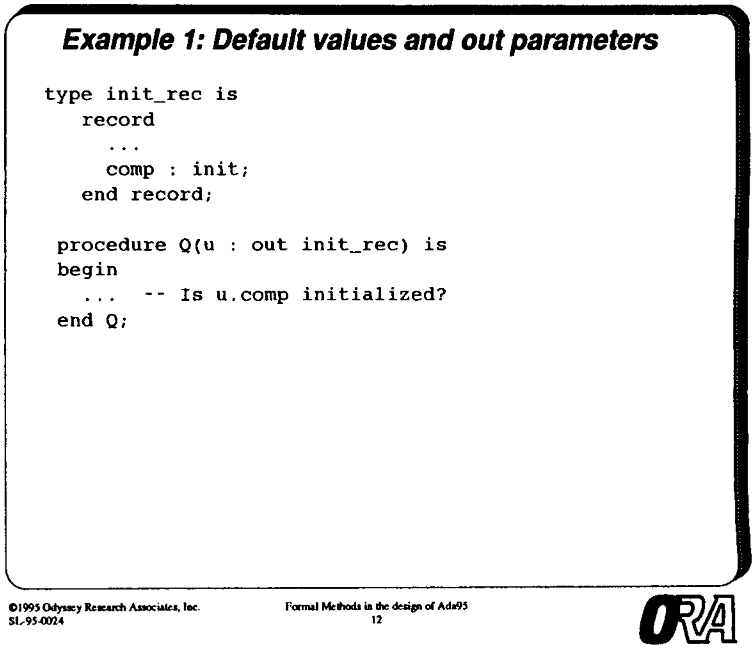
## Opportunism in action: default values

- □ Proposal to permit default values for subtypes

```
subtype init is integer range 1 .. 100 := x+6;
y : init;      -- initialized to x+6;
z : init := 3; -- initialized to 3;
```

- □ One goal, presumably is the following invariant:
  - ○ Objects of subtype `init` will always have a defined value.

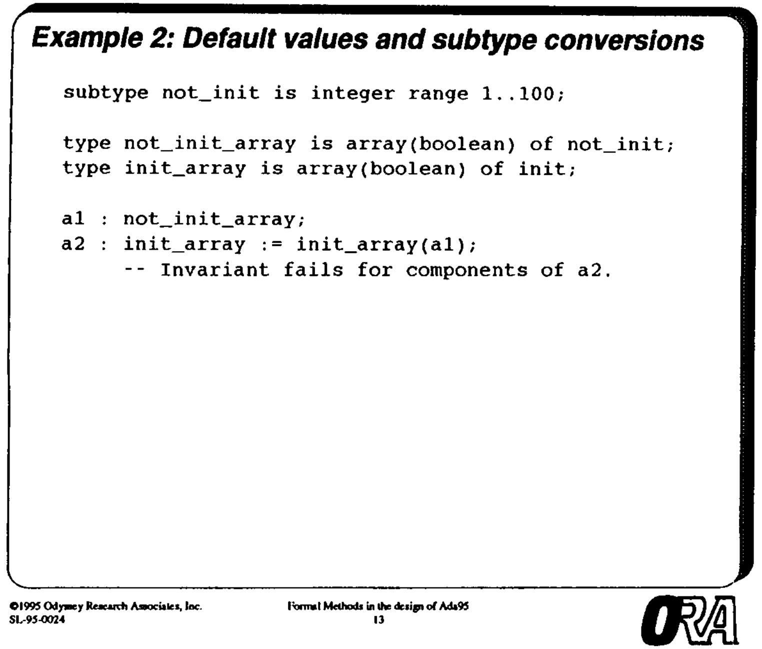
SL-95-0024



## Example 1: Default values and out parameters

```
type init_rec is
   record
     ...
     comp : init;
   end record;

 procedure Q(u : out init_rec) is
 begin
   ...  -- Is u.comp initialized?
 end Q;
```

SL-95-0024

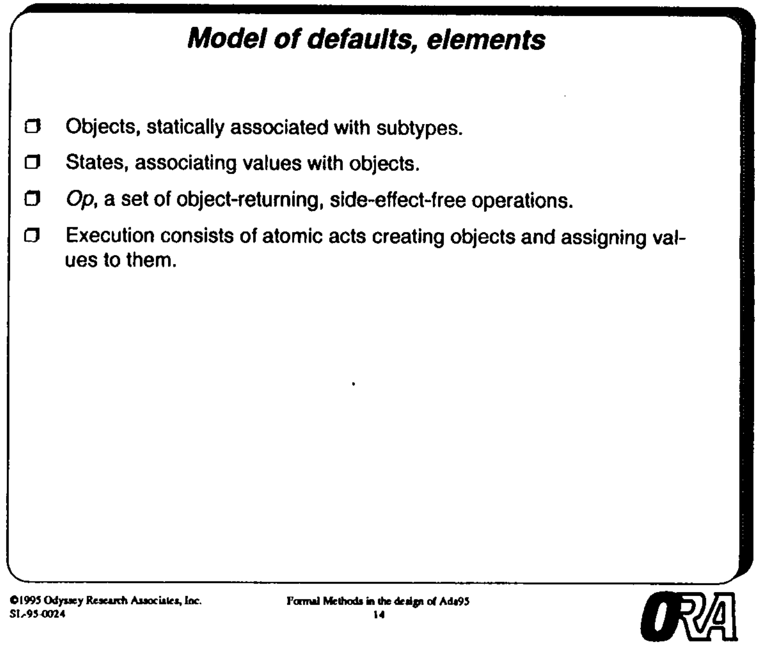


## Example 2: Default values and subtype conversions

```
subtype not_init is integer range 1..100;

type not_init_array is array(boolean) of not_init;
type init_array is array(boolean) of init;

a1 : not_init_array;
a2 : init_array := init_array(a1);
     -- Invariant fails for components of a2.
```

©1995 Odyssey Research Associates, Inc.
SL-95-0024

Formal Methods in the design of Ada95
13

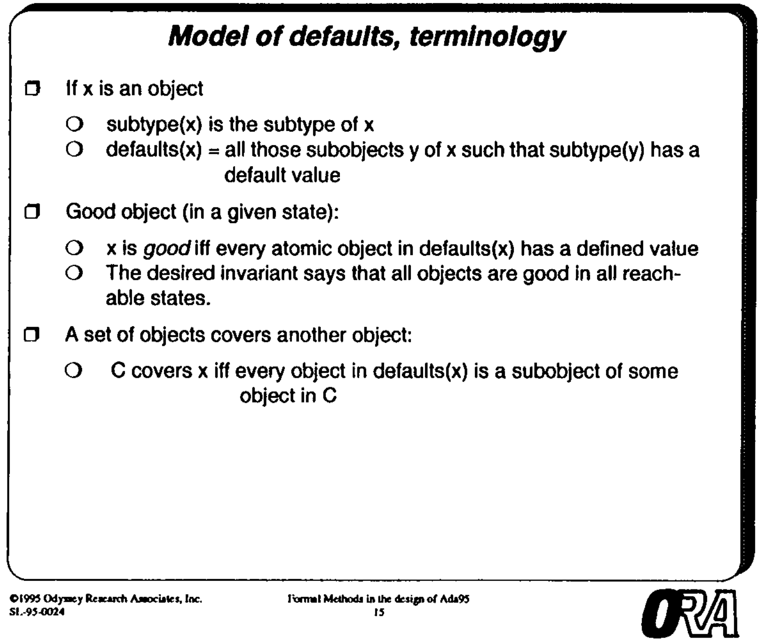


## Model of defaults, elements

- ❒ Objects, statically associated with subtypes.
- ❒ States, associating values with objects.
- ❒ *Op*, a set of object-returning, side-effect-free operations.
- ❒ Execution consists of atomic acts creating objects and assigning values to them.

©1995 Odyssey Research Associates, Inc.
SL-95-0024

Formal Methods in the design of Ada95
14

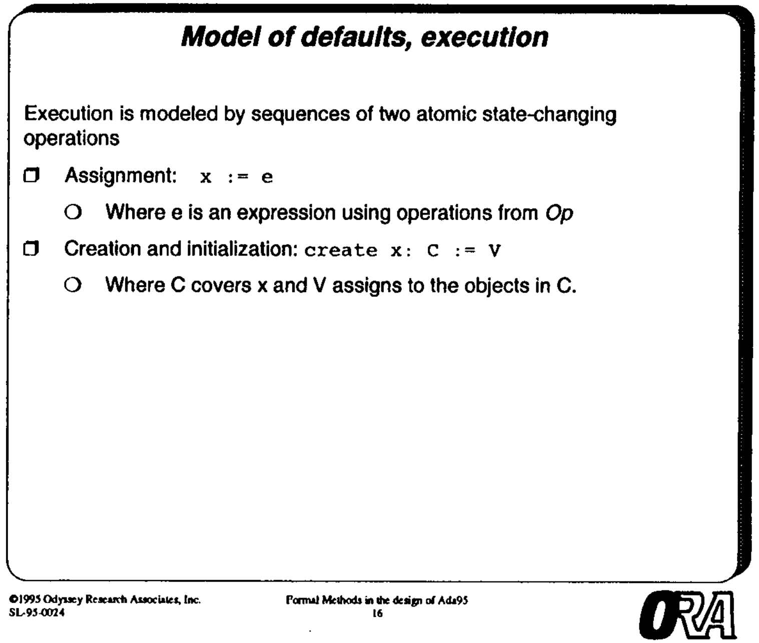


## Model of defaults, terminology

- ❒ If x is an object
  - ○ subtype(x) is the subtype of x
  - ○ defaults(x) = all those subobjects y of x such that subtype(y) has a default value
- ❒ Good object (in a given state):
  - ○ x is *good* iff every atomic object in defaults(x) has a defined value
  - ○ The desired invariant says that all objects are good in all reachable states.
- ❒ A set of objects covers another object:
  - ○ C covers x iff every object in defaults(x) is a subobject of some object in C

©1995 Odyssey Research Associates, Inc.
SL-95-0024

Formal Methods in the design of Ada95
15



## Model of defaults, execution

Execution is modeled by sequences of two atomic state-changing operations

- ❒ Assignment: `x := e`
  - ○ Where e is an expression using operations from *Op*
- ❒ Creation and initialization: `create x: C := V`
  - ○ Where C covers x and V assigns to the objects in C.

©1995 Odyssey Research Associates, Inc.
SL-95-0024

Formal Methods in the design of Ada95
16

## Model of defaults, example axioms

Axioms describe how state-changers affect the goodness of objects

- ❐ `y := e`
  - ○ If y is a subobject of x, x is good, and e is good, then after this execution x is still good.
- ❐ `create x: C := V`
  - ○ If the values assigned by V are good then, after this execution, x is good.
- ❐ Etc.

Immediate theorem:

> The invariant is guaranteed if the operations in *Op* are non-decreasing: good arguments yield a good result under normal termination.

## To apply the model:

Can the proposed language rules be described within the given model of defaults? If so, the invariant holds.

- ❐ Revisit example 1 (out parameters): How is an out parameter created?
- ❐ Revisit example 2 (subtype conversions): Is subtype conversion a non-decreasing operation?

## Default subtypes, concluded

- ❐ Proposal dropped
- ❐ Not a "point change"

## Overload resolution

- ❐ Ada83 rules are complex
  - ○ Unpleasant surprises
  - ○ Fine points not consistently interpreted by compilers
- ❐ Proposed changes add complexity (in some ways)
- ❐ Problems describing proposed changes
- ❐ Incompatibilities between old rules and proposed changes unclear

## Problems

- □ Finding satisfactory rules for overload resolution
  - ○ Interactions of features:
    - implicit conversion
    - OOP
    - strong typing
    - overload resolution
  - ○ Special role of operator symbols
    - Proposal for "primitive visibility"
  - ○ Blemishes in proposals
    - non-local
    - upward inconsistent
    - "Beaujolais effects"
- □ Describing the rules you've found
  - ○ Descriptions of proposals (both declarative and algorithmic) flawed

## Hierarchical names and "use" clauses

```
 package p is
  function g return foo;
  function "+"(x,y: foo) return bar;
 end package:
```

Compare

```
 x : foo := P.g;
 b : bar := P."+"(e1, e2);
```

with

```
 use p;
 x : foo := g;
 b : bar := e1+e2;
```

## Beaujolais effect

```
 package P is
    function f(a: boolean) return boolean; -- F#1
end P;

function f(a: integer) return boolean;     -- F#2
function "<"(a: integer; b: integer) return integer;

[-- use P;]
x : boolean := f(1 < 2);    -- F#1 or F#2?
```

## Abstract formal model of overload resolution

(by Bill Easton)

- □ Written in Z
- □ Parameters to model:
  - ○ The environment (what declarations apply)
  - ○ Which interpretations are preferred
- □ Hides irrelevant detail
  - ○ Makes concrete syntax abstract (e.g., all expressions become applications)
  - ○ Omits specs of irrelevant operations (e.g., the check that actuals match formals)
  - ○ Applies to one expression at one point in program (environment is not a state variable)

## Using the formal model of overload resolution

Different rules correspond to different choices of parameters

For particular rules, we may ask

- □ Are the rules "local"?
- □ Are there Beaujolais effects?
- □ Are there upward inconsistencies?
- □ What are the upward incompatibilities?

## Results of work on overload resolution

- □ Understood
  - ○ Formalism chosen to fit the problem
- □ Received with enthusiasm
  - ○ Shed light on complex problem
- □ Suggested modifications adopted by MRT

Successful because

- □ Addresses acknowledged difficulty
- □ Suitable problem (isolatable)
- □ Right abstraction
- □ Right problem solver (mathematician and Ada lawyer)

## Object oriented programming

Ada9X inheritance: generalizes type derivation

```
 with Calendar;
package alert_system is

   type alert is tagged
     record
       Time_Of_Arrival: Calendar.Time;
       Message: Text;
     end record;

   procedure Display(A: in Alert; ...);
   procedure Handle(A: in out alert);
   ...

   type medium_alert is new alert with
     record
       Action_Officer: Person;
     end record;

   ...

 end alert_system;
```

## Classwide types:

The type Alert'class

- □ like a variant record type
- □ tag acts as discriminant
- □ alternatives are the descendants of Alert

## Ada9X runtime binding:

```
procedure Process_Alerts(AC: in out Alert'Class) is
begin
   ...
   Handle(AC);    -- dispatch according to tag
   ...
end;
```

## Goal of work on OOP:

□ Proof formalism for a significant subset of the OOP constructs

□ Extend existing formalism (Penelope) for Ada83

Interacting with the MRT

□ Is the model right?

□ Does the model cover enough?

Difficulties:

□ OOP semantics is a research topic

□ Ada9X OOP proposals very volatile

□ Many non-semantic issues (syntax, static semantics, efficiency)

□ LPT is an add-on

## Conclusions of OOP report

□ Proof formalism and specification notation for basic OOP mechanisms

□ Justifies claim that semantics is clean

□ Practical? Only applied to simple examples

## Phase 2, approach

□ Ignore static semantics

□ Define dynamic semantics in "natural semantics" formalism

□ Written in notation of Typed Prolog

- ○ static semantic checking
- ○ definition executable on small examples

## Phase 2, summary

□ Several problems found, corrected in final reference manual

□ Surprises: difficulties in seemingly "trivial" areas

## Conclusions: the Language Precision Project

□ Opportunistic approach achieved some influence on the MRT

- ○ Some results unlikely to come from a different source

□ Begun to identify a verifiable subset

□ Limited by not being part of the MRT

- ○ Hard to find the best assignments
- ○ Not always in the loop
- ○ Contractual requirement for "products" not ideal

□ Widely scattered LPT manageable, but awkward

□ Differences in culture

- ○ LPT, semantics; MRT, implementation
- ○ Key demand on members of LPT – bridge the gap

Hope: work like this will become standard operating procedure

# INTRODUCTION TO PENELOPE

***David Guaspari***

***Odyssey Research Associates***

A formal program verification is a (mathematical) proof that a program executed according to its intended model meets some specification. This proves that the algorithm defined by the program is *correct* in the precise technical sense of being consistent with a particular specification. A program correct in this sense is free from a large and important class of errors, even though its behavior may still produce unintended results---either because the implementation of the programming language itself does not match the model of execution, or because the specification does not correctly express the user's intentions.

Penelope is a prototype system for interactively developing and verifying programs that are written in a rich subset of sequential Ada. Penelope can be used to develop a program and its correctness proof incrementally, and in concert with one another. Incrementality is used in a number of ways to help make verification more tractable and more productive. For example, if an already-verified program is modified, one can attempt to prove the modified version by replaying and modifying the original verification.

Penelope's specification language, Larch/Ada, belongs to the family of Larch interface languages. Larch/Ada scales up properly, in the sense that it is demonstrably sound to decompose a system hierarchically and reason locally about the implementation of each piece.

Penelope has been applied in various demonstration projects---for specification (guidance control, distributed operating system), verification (of off-the-shelf code), and formal development (by non-expert as well as expert users). Some features of Penelope have been embodied in AdaWise, a lint-like non-interactive tool that warns of the potential for certain dynamic semantic errors in Ada programs.

# Introduction to Penelope

**_David Guaspari_**

**_NASA Formal Methods Workshop_**
**_May 10 - 12, 1995_**

**_Odyssey Research Associates_**
**_301 Dates Drive_**
**_Ithaca, NY 14850-1326_**
**_(607) 277-2020_**
**_davidg@oracorp.com_**

## Goals of the Penelope project

- Define the semantics of a large Ada subset
- Define an Ada specification language (Larch/Ada)
- Implement automated support for reasoning about specifications and implementation (Penelope)
- Apply Penelope

## Results

- Mathematical goals met
- Penelope supports a non-trivial subset of our model
- Applications have been demonstrations
- Spin-offs
  - "Lightweight" Ada tools (AdaWise)
  - Carryover to Ada95 work (LPT)
  - Penelope as a teaching tool

## Outline of this talk

- Background (modeling Ada)
- The Larch/Ada specification language
- The Penelope system
- Some Penelope and AdaWise applications

## Semantics of Ada

□ Complicated parts

○ static semantics
○ concurrency

□ Dynamic semantics of sequential Ada is clean

○ strong typing
○ run-time constraint checking
○ disciplined use pointers
○ disciplined use of exceptions
○ information hiding
○ standardizes semantics often left implementation-dependent

□ Dark corners of sequential Ada:

○ arbitrary choices left underdetermined
○ interactions of optimization and exceptions

## Definitions of Ada

□ In the Ada culture

○ Officially: LRM + AI's
○ Ad hoc standard: ACVC

□ Formal definitions

○ Formal definition of sequential Ada80
○ AdaEd interpreter (in SetL)
○ DDC + CRAI definition of Ada83 (virtually complete)

□ Formal definitions of subsets

○ ORA
○ CLInc
○ Aerospace Corporation
○ Program Validation, Ltd.
○ ProSpectra

## Modeling our Ada subset

□ Eliminate almost all dark corners by static semantic checks

○ for "improper" aliasing
○ for "undisciplined" side effects

□ Disallow optimizations sanctioned by RM 11.6.

□ Ignore resource limitations (storage error, numeric overflow).

Definitional technique: denotational semantics, predicate transformers.

## Subset covered by the model

□ Nearly all non-pathological uses of the following:

○ All sequential types (including floating point)
○ All sequential statements
○ Exception-raising and -handling
○ Subprograms (including side-effects, recursion, global variables)
○ Packages
○ Generics

## What a Penelope proof proves

- ❒ Hypotheses on compiler
  - ○ No section 11.6 optimizations
  - ○ No optional program_error
- ❒ Hypotheses on executions
  - ○ No storage error
  - ○ No numeric overflow
- ❒ Currently unchecked
  - ○ Consistency conditions on theories

For allowed executions on allowed compilers: if initial conditions satisfied then termination implies exit conditions satisfied.

## Predicate transformer semantics: a "logical" form of symbolic execution

```
--|      ? --(1) precondition (sought)
x := x+1;
--| y < x --(2) postcondition (given)
```

Question: What must be true at (1) to guarantee "y < x" will be true at (2)?

Answer: "y < x+1"

Answer obtainable by symbolic manipulation: substitute "x+1" for "x" in "y < x"

Analogy between predicate transformers

```
wlp(S,__): predicate->predicate
```

which take postcondition to precondition, and continuation semantics

```
M[[S]]: continuation -> continuation
```

which takes (post)continuation to (pre)continuation.

This connection is developed in work by Wolfgang Polak.

Systematically associate computational entities with symbolic entities, e.g.:

| Computational | Symbolic |
|---|---|
| value | term |
| answer | {true, false} |
| continuation | predicate |

Derived predicate transformer semantics can be proven sound for the corresponding denotational model.

## Penelope supports a special "asymptotic" floating point semantics

- ❐ Models computation in the limit as machine accuracy improves
- ❐ Purely algebraic reasoning about approximate operations – approximate equality, etc. (no epsilons and deltas)
- ❐ Highlights logical errors in specifying and coding with discontinuous operations (such as floating point comparisons)

## Larch/Ada

- ❐ Specifies sequential Ada
  - ○ Assertional (entry-exit, invariants, ...)
  - ○ Partial correctness
- ❐ Unit of specification is the compilation unit
- ❐ Annotations of bodies hidden from clients
- ❐ "Two-tiered" semantics

## Two-tiered specifications:

- ❐ Mathematical component
  - ○ An environment of mathematical definitions
- ❐ Interface component
  - ○ Describes behavior in terms of environment

## Informal example of a two-tiered specification

```
function plus(x,y: integer) return
   integer;
```

- ❐ Mathematical component
  - ○ Defines sort Int, the infinite collection of mathematical integers
  - ○ Defines the basic mathematical operations on Int (+, *, <, ...)
- ❐ Interface component
  - ○ Type integer is *based on* sort Int
  - ○ Behavior of plus
    - ▮ Entry: No assumptions about state of parameters
    - ▮ Normal exit: Return x+y (mathematical sum), if termination is normal
    - ▮ Exceptional exit: raise error iff, on entry `x+y < -(2`$^{32}$`)` or `x+y > 2`$^{32}$` - 1` (all operations mathematical)
    - ▮ No side effects

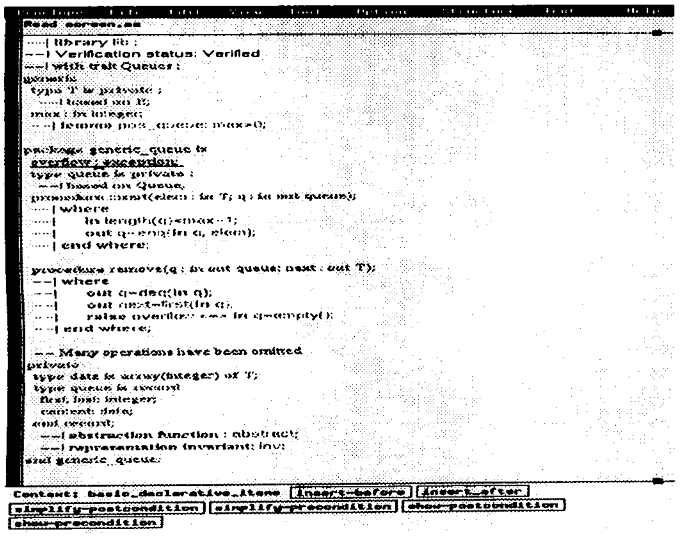
## Mathematics defined in Larch Shared Language (LSL)

- ❒ Designed by John Guttag (MIT), Jim Horning (DEC SRC)
- ❒ Notation for ordinary mathematics
  - ○ Sorts (sets of values)
  - ○ Operations on sorts
  - ○ Logical operations
  - ○ Strongly sorted (i.e., typed)
- ❒ "Trait" – A modular way to describe theories
  - ○ axioms
  - ○ assumptions
  - ○ conclusions
- ❒ Independent of program language

We have extended Larch

## The Penelope System

- ❒ Editor for Ada programs with Larch/Ada annotations
- ❒ Automated support for developing (extended) Larch Shared Language
- ❒ Automated support for assertional reasoning about code
  - ○ Incrementally generates verification conditions (VCs)
  - ○ VCs are statements in ordinary logic
  - ○ Supports Dijkstra's goal-directed style of program development
- ❒ Permits incremental reasoning
  - ○ Factors proofs
  - ○ Proofs are replayable
- ❒ Simple library, pretty-printing
- ❒ Implemented with the Synthesizer Generator



```
Read screen.aa
---| library lib ;
--| Verification status: Verified
--| with trait Queue :
generic
  type T is private ;
   --| based on E;
  max : in integer;
  --| lemma pos_queue: max>0;

package generic_queue is
  overflow : exception;
  type queue is private ;
   --| based on Queue.
  procedure insert(elem : in T; q : in out queue);
  --| where
  --|     in length(q)<max-1;
  --|     out q=enq(in q, elem);
  --| end where;

  procedure remove(q : in out queue; next : out T);
  --| where
  --|     out q=deq(in q);
  --|     out next=first(in q).
  --|     raise overflow <=> in q=empty();
  --| end where;

  -- Many operations have been omitted
private
  type data is array(integer) of T;
  type queue is record
   first, last: integer;
   content: data;
  end record;
   --| abstraction function : abstract;
   --| representation invariant: inv;
end generic_queue;
```

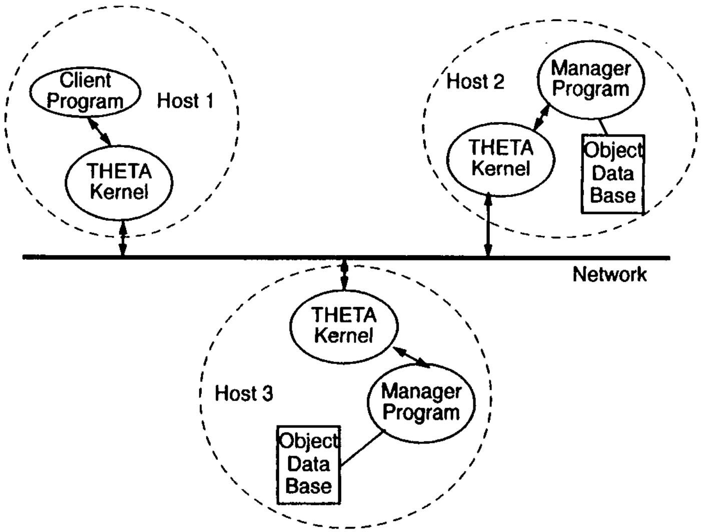
Content: basic_declarative_items

## Some applications

- ❒ Theta kernel (security, specification exercise)
- ❒ COTS code (calendar package)
- ❒ Use by non-expert user (generic sets package for 777)
- ❒ Applications of AdaWise (to itself, Theta, repository code)
- ❒ Orbit calculations (floating point)
- ❒ Specification exercises (guidance control, flight management)

## The Theta kernel

- Authentication and message routing for a trusted distributed heterogeneous operating system
- Penelope used to record high-level design, perform simple refinement proofs
- Described in "Applications of Formal Methods", edited by M.G. Hinchey and J.P. Bowen, Prentice Hall International Series in Computer Science, Hemel Hempstead, 1995.



## COTS code (calendar package)

- Undocumented assumptions
  - No impossible dates are entered
  - Strings are numbered from 1
- Years represented by last two digits
- Careless arithmetic -
  - E.g., `Natural((day mod 7) - 1)`vs.
    `Natural((day - 1) mod 7)`

## Non-expert user (generic sets package)

- User expert in testing
- Training:
  - Four days of supervised practice (at ORA)
  - Regular e-mail exchanges
- Difficulties
  - The "usual" FM problems
  - Penelope is not robust
  - Currently supported subset required workarounds
- User succeed in specifying and proving generic sets package (roughly 6 weeks of work from scratch)

## AdaWise -- "Lightweight tools"

- □ Push-button (lint-like)
- □ Warns of potential for certain run-time problems
- □ Conservative (no warning implies no problem)
- □ Built on ASIS (runs with RISCAda, SunAda, Rational Apex)

Funded by Air Force Rome Laboratories and ARPA (STARS)

## Properties checked

- □ Improper aliasing
- □ Incorrect order dependence
- □ Access to undefined scalars (under development)

## Experience (Theta, repository code)

- □ Elaboration order dependences are common (mostly a portability problem)
- □ Obscure aliasing problems fairly common
- □ False warnings often point to doubtful coding practices
- □ Size of individual examples: up to 50-60 modules, 18,000 SLOC

## Alias Warning: Non-Scalar Parameters

- □ Forms Generator:

```
**** form_executor lines  128 to  130:
            FORM_MANAGER.GET_FIELD_INFO
                  (FIELD, NAME, POSITION, LENGTH, RENDITION,
                  CHAR_LIMITS, VALUE, VALUE, MODE)

>> Parameters:  7 and  8 are potential ALIASES
>> (potential ORDER of COPY OUT error) and
>> (potential ERRONEOUS EXECUTION)
```

## Alias Warning: Non-Scalar Parameters (cont'd)

```
procedure GET_FIELD_INFO( ... ;
                          INIT_VALUE : out FIELD_VALUE;
                          VALUE : out FIELD_VALUE;
                          ...) is
begin
   ...
   VALUE := FIELD.VALUE;
   MODE := FIELD.MODE;
exception
   ...
end GET_FIELD_INFO;
```

## Further Work

- ❒ Tractability of constraint checking, definedness checking
- ❒ Packages with state
- ❒ Specification of termination proofs
- ❒ Improved modularity
- ❒ Full support for generics
- ❒ Concurrency

omit

# Session 6: Hardware Systems

**Paul Miner, Chair**

- **The Formal Verification Technology Used on AAMP5**, by *Mandayam Srivas*, SRI International
- **Specification and Verification of VHDL Designs**, by *Damir Jamsek*, Odyssey Research Associates
- **Derivational Reasoning System**, by *Bhaskar Bose*, Derivation Systems Inc.

# The Formal Verification Used for the AAMP5 and AAMP-FV

It is becoming increasingly evident within the VLSI design industry that the complexity of many current hardware designs is outstripping the capability of traditional simulation-based tools to adequately verify them. This situation was well-illustrated by the recent floating point bug discovered in Intel's Pentium processor. The industry is beginning to look at formal verification as a technological alternative to simulation for obtaining higher assurance than is currently possible.

Recently, SRI International and Collins Commercial Avionics, a division of Rockwell International, undertook a project to explore how formal techniques for specification and verification could be introduced into an industrial process. The project, sponsored by the Systems Validation Branch of NASA Langley and Collins Commercial Avionics, consisted of specifying in the PVS language a portion of a Rockwell proprietary microprocessor, the AAMP5, at both the instruction set and register-transfer levels and using the PVS interactive proof-checker to show that the microcode correctly implemented the specified behavior for a representative subset of instructions.

The main goal of the project was two-fold: First, to investigate the feasibility of formally specifying and verifying a complex commercial microprocessor that was not expressly designed for formal verification. Second, to explore effective ways to transfer the technology to an industrial setting. The choice of the AAMP5 satisfied the first goal since the AAMP5 was not designed for formal verification, but to provide a more than threefold performance improvement while remaining object-code-compatible with the earlier AAMP2, which is used in numerous avionics applications, including the Boeing 737, 747, 757, and 767.

To satisfy the technology transfer objective, we had to develop a suitable verification methodology and a formal infrastructure to make the technology usable by practicing engineers. This infrastructure includes techniques for decomposing the microprocessor verification problem into a set of verification conditions that the engineers can formulate and strategies to automate the proof of the verification conditions. The development of the infrastructure was one of the key accomplishments of the project. Most of the infrastructure and methodology are general enough to be reused for other microprocessors, certainly in the verification of another member of the AAMP family. This methodology was used to formally specify the entire microarchitecture and more than half of the instruction set and to verify a core set of eleven AAMP5 instructions representative of several instruction classes. However, the methodology and the formal machinery developed are adequate to cover most of the remaining AAMP5 instructions. Although PVS was the vehicle of the experiment, the methodology is applicable to other sufficiently powerful theorem provers.

Another key result of the project was the discovery of both actual and seeded errors. Two actual microcode errors were discovered during development of the formal specification, illustrating the value of simply creating a precise specification. Both were specific to the AAMP5 and were corrected before first fabrication. Two additional errors seeded by Collins in the microcode were systematically uncovered by SRI, who knew that bugs had been seeded, but not their location or identity, while doing correctness proofs. One of these was an actual error that had been discovered by Collins after first fabrication but left in the microcode provided to SRI. The other error was designed to be unlikely to be detected by walk-throughs, testing, or simulation.

Steve Miller's talk earlier in the workshop, gave an overview of the AAMP5 project emphasizing the technology transfer process with its administrative and managerial aspects. This talk describes the technical approach used in verifying the AAMP5. Please refer to Steve Miller's slides for the AAMP5 design figures.

# The Formal Verification Technology Used for the AAMP5 and AAMP-FV*

Mandayam Srivas

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
(srivas@csl.sri.com)
(http:\\www.csl.sri.com)

*Joint work with Steve Miller, Rockwell International, Collins Commercial Avionics, Cedar Rapids, Iowa 52498

## Organization of the Talk

- Overview of AAMP5 Processor
- What did we verify?
- How did we manage complexity of verification?
- Mechanization of proofs
- Conclusions

## The Stack Memory Model

## Macromachine Specfication

```
ADD_lemma_1: LEMMA
  LET X = current_data_env(st)(tos(st)+1),
      Y = current_data_env(st)(tos(st))
  IN current_opcode(st) = ADD & NOT arith_update.exception(st) =>
     normal_macro_machine.next_macro_state(st) =
       st WITH [(dmem)(word2denv(denv(st)))(tos(st)+1) := X + Y,
                (pc)  := pc(st)  + 1,
                (tos) := tos(st) + 1]
```

## DPU Environment Assumptions

- "If RDY is false then RDY will eventually become true provided the DPU obeys its end of the protocol."

```
CLR_RDY_wait: AXIOM
   NOT RDY(t) & normal_fetching(t) & CLR_RDY?(NCO(t)) =>
      (EXISTS (t1 | t1 > t):
          stays_same(NCO(t))(t, t1)  =>
            stays_same(RDY)(t, t1-1) & RDY(t1) &
               normal_fetching(t1) &
                 DP(t1) = DP(t)+length_of_instruction(opcode) )
```

## DPU Environment Assumptions

- "If RDY is true then decoded outputs presented by the LFU correspond to the instruction stored at the PC presented by the LFU."

```
% Invariant constraints on the outputs when LFU is RDY
RDYinv: AXIOM
   RDY(t) =>
     LET opcode = opcode_at(CENV(t), DP(t), CODE_MEMORY(t))
     IN  EP(t) = EP_ROM(opcode) & (has_imm_data(opcode)
           => IM(t) = first_unit_of_imm_data(t))
```

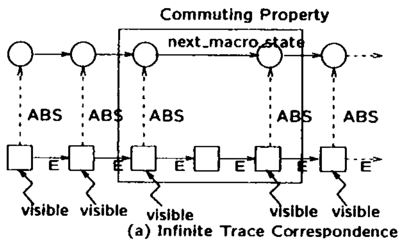
## Sample Register-Transfer Level Specification

```
SVax: AXIOM SV(t+1) = IF DHLD(t) THEN SV(t) ELSE NEXT_SV(t) ENDIF

SV1ax: AXIOM SV1(t+1) = IF DHLD(t) THEN SV1(t) ELSE SV(t) ENDIF

TVax: AXIOM TV(t+1) = IF DHLD(t) THEN TV(t) ELSE NEXT_TV(t) ENDIF

TV1ax: AXIOM TV1(t+1) = IF DHLD(t) THEN TV1(t) ELSE TV(t) ENDIF

END SVTVpipeline_axioms
```

## Other Microprocessor Verification Efforts

- FM8501 [Hunt: 1986]
- Viper [Cohn: 1988]
- Tamarack [Joyce: 1988]
- MiniCayuga [Srivas & Bickford: 1990]
- Saxe's Pipeline [Saxe, et. al. 1991]
- DLX pipeline [Burch & Dill: 1993]
- UNITA [Windley: 1994]
- ...

## General Microprocessor Correctness



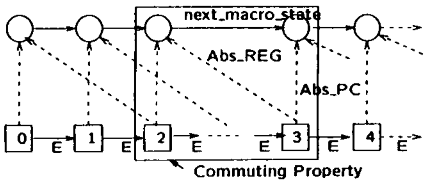
(a) Infinite Trace Correspondence

- Abstraction function (ABS)
- Visible state

## Complications posed by AAMP5

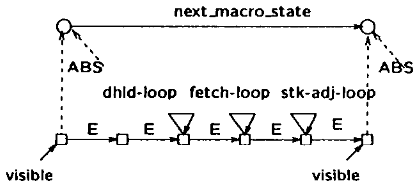
- Pipelining
- Autonomous prefetch and data transfers
- The "stack cache" abstraction

## Pipelined Microprocessor Correctness



- Abstraction function must be suitably "skewed"
- Length of instruction cycle can be idefinite not necessarily a function of the current visible state

## AAMP5 Pipeline Correctness



## Specializing Correctness Criterion to the AAMP5

- Visible state:
  - The first microinstruction of the *current* macroinstruction is loaded into MC0 (microregister)
  - The current (compute-stage) instruction is guaranteed to advance
  - The previous (write-stage) instruction in the pipeline is guaranteed to complete in the next cycle
- Abstraction function
  - A projection function except for data memory
  - The data memory at the macro level is real memory with the overlaid "stack registers"

## Formal Correctness Statement

```
commuting_property: LEMMA
  visible_state(t) & proper_instrns_in_pipe(t)
          & no_logical_stack_overflow(t)
    => EXISTS (tp: time | tp > t):
          stays_low(t+1, tp-1)(visible_state) &
             visible_state(tp) & ABS(tp) = next_macro_state(ABS(t))

visible_state(t): bool =
  MC0(t) = ROM(zero_extend[8](11)(EP_ROM(current_op(t))(t))) &
     & NOT(DHLD(t)) & NOT(nextDJMP(t))
         & entry_cond_met(current_op(t))(t)
```

## Our Approach to the Verification Task

- Incremental verification: Structure it so as to
  - get early feedback
  - Collins staff can participate in the proof process
- Abstract the DPU environment: A set of "rely-guarantee" assertions characterizing the interactions for the DPU with the other units.
- Decompose the proof into
  - an automatic part: *instruction-specific verification conditions*
  - interactive part: *general verification conditions*

## General Verification Conditions

- `dhld_lemma`:
  "NOT DHLD ⇒
  ◇ (DHLD & macrostate unchanged)"
- `LFU_not_rdy_lemma`:
  "NOT RDY ⇒
  ◇ (RDY & next instruction moves in)"
- `stack_adjust_lemma`:
  "NOT entry_condition_met ⇒
  ◇ (entry_condition_met & macrostate unchanged"

## Instruction-specific Verification Conditions

```
TO_correctness: LEMMA
   visible_state_with(ADD)(t) & SysInv(t) =>
      TO(t+2) =
        (sign_extend[16](48)(ith_top_of_scache(t+1)(1))
          + sign_extend[16](48)(ith_top_of_scache(t+1)(0)))^(15,0)

i: VAR below[10]
REG_correctness: LEMMA
   visible_state_with(ADD)(t) & SysInv(t) =>
                  REG(t + 2)(i) = REG(t + 1)(i)
```

## Mechanization of proofs of Verification Conditions

- Overview of PVS
- Our core hardware strategy

## The Core Startegy Used

For proving theorems of the form:
"Precond(s0) $\Rightarrow$ $f(s0) \;=\; g(E^i(s0))$, for some constant i"

- Rewrite expressions in the theorem using the functions in the design specification as auto-rewrite rules.
  `((repeat (do-rewrite)))`
- "Lift" all the if-then-else structures to the topmost level
  `((repeat (lift-if)))`
- Simplify using a BDD simplifier, arithmetic, and other decision procedures along with using a library of bit-vector properties as auto-rewrite rules.
  `((then* (bddsimp) (assert)))`

## Mechanization overview

- Instruction-specific VCs: Core hardware strategy
- General VCs: Inductions with core strategy for the base and inductive steps.
- Main correctness theorem: Chaining of VCs with intruction-specific VCs, definition of abstraction function, and macro specification used as rewrite rules.

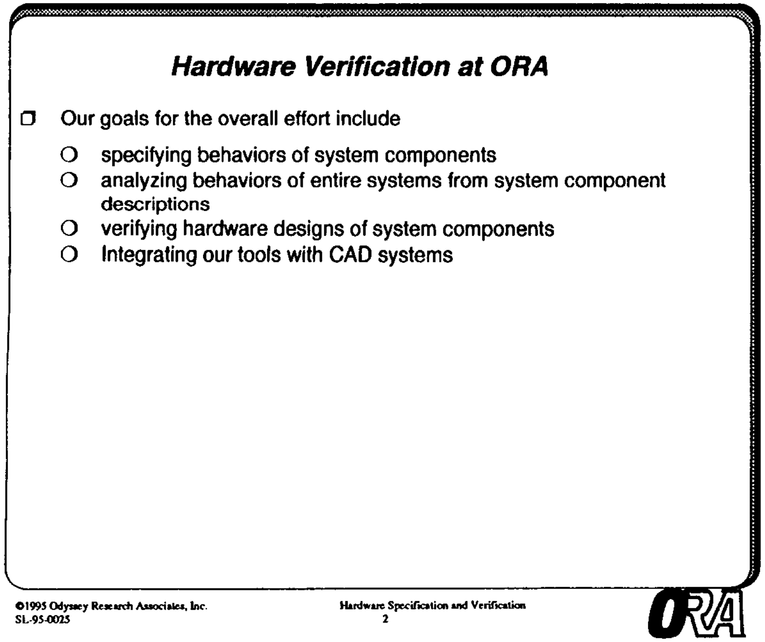
## Conclusions

- Is commercial processor verification technically feasible?
  Yes, if carefully planned and executed.
- Can practicing engineers use this technology?
  Yes, with appropriate training and expert help.
- Is it "cost-effective" ?

omit

# Specification and Verification of VHDL Designs

Damir Jamsek
Odyssey Research Associates
May 11, 1995



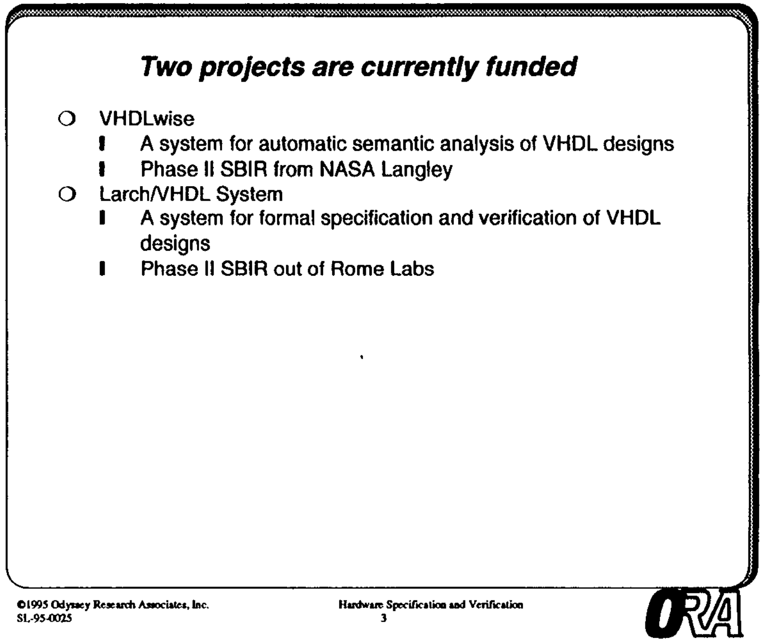
## Hardware Verification at ORA

❒ Our goals for the overall effort include

- ○ specifying behaviors of system components
- ○ analyzing behaviors of entire systems from system component descriptions
- ○ verifying hardware designs of system components
- ○ Integrating our tools with CAD systems

©1995 Odyssey Research Associates, Inc.
SL-95-0025

Hardware Specification and Verification
2

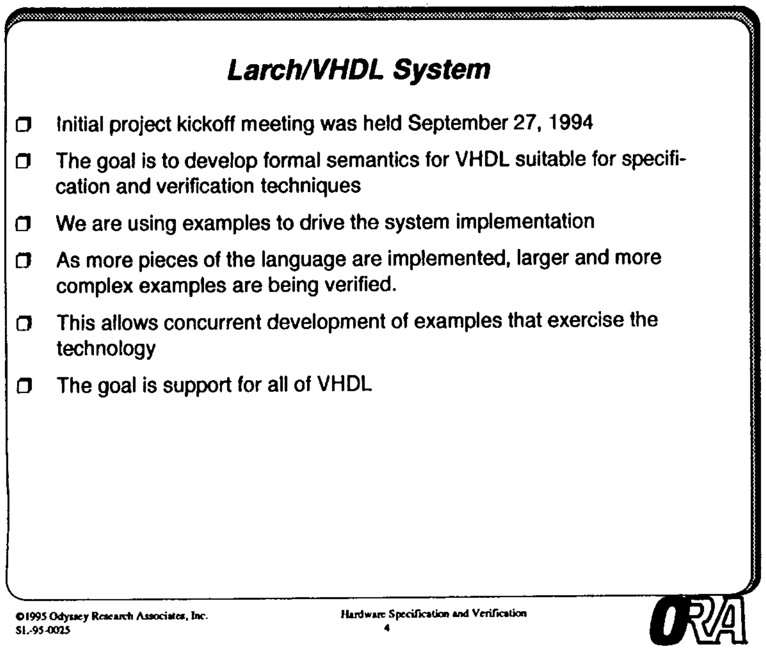


## Two projects are currently funded

- ○ VHDLwise
  - ▮ A system for automatic semantic analysis of VHDL designs
  - ▮ Phase II SBIR from NASA Langley
- ○ Larch/VHDL System
  - ▮ A system for formal specification and verification of VHDL designs
  - ▮ Phase II SBIR out of Rome Labs

©1995 Odyssey Research Associates, Inc.
SL-95-0025

Hardware Specification and Verification
3



## Larch/VHDL System

❒ Initial project kickoff meeting was held September 27, 1994

❒ The goal is to develop formal semantics for VHDL suitable for specification and verification techniques

❒ We are using examples to drive the system implementation

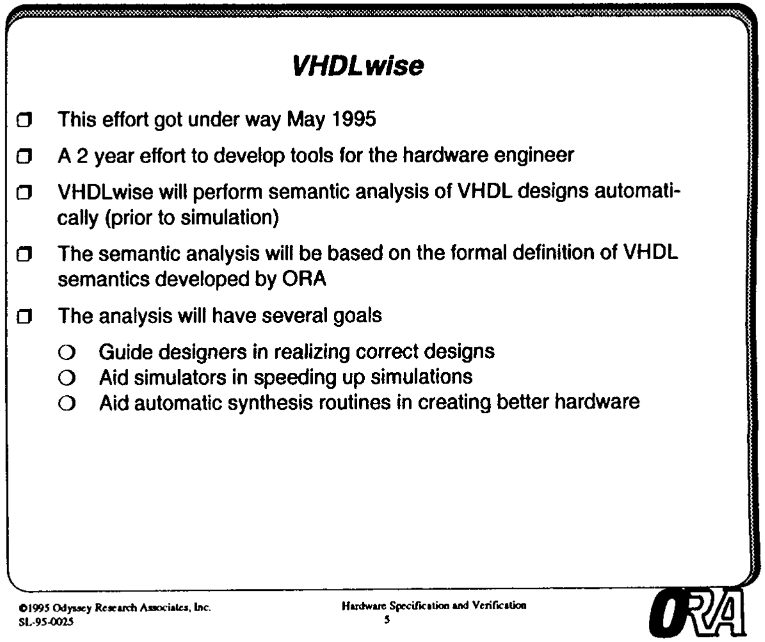
❒ As more pieces of the language are implemented, larger and more complex examples are being verified.

❒ This allows concurrent development of examples that exercise the technology

❒ The goal is support for all of VHDL

©1995 Odyssey Research Associates, Inc.
SL-95-0025

Hardware Specification and Verification
4



## VHDLwise

❒ This effort got under way May 1995

❒ A 2 year effort to develop tools for the hardware engineer

❒ VHDLwise will perform semantic analysis of VHDL designs automatically (prior to simulation)

❒ The semantic analysis will be based on the formal definition of VHDL semantics developed by ORA

❒ The analysis will have several goals

- ○ Guide designers in realizing correct designs
- ○ Aid simulators in speeding up simulations
- ○ Aid automatic synthesis routines in creating better hardware

©1995 Odyssey Research Associates, Inc.
SL-95-0025

Hardware Specification and Verification
5

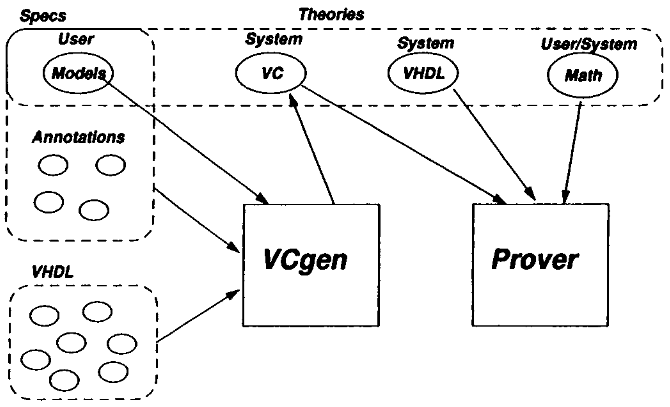
## VHDLWise Development

- ❒ Phase I developed a basis for semantic analysis of VHDL designs prior to simulation
- ❒ Algorithms based on formal semantics of VHDL will automatically check for presence or absence of certain kinds of properties in VHDL designs.
- ❒ Examples of such properties include
  - ○ Wait loop checker
  - ○ Cancelled Signal Checker
  - ○ UninitializedVariable Checker
- ❒ Each kind of check may aid in one of three ways
  - ○ Guide designers in realizing correct designs
  - ○ Aid simulators in speeding up simulations
  - ○ Aid automatic synthesis routines in creating better hardware

## Larch/VHDL

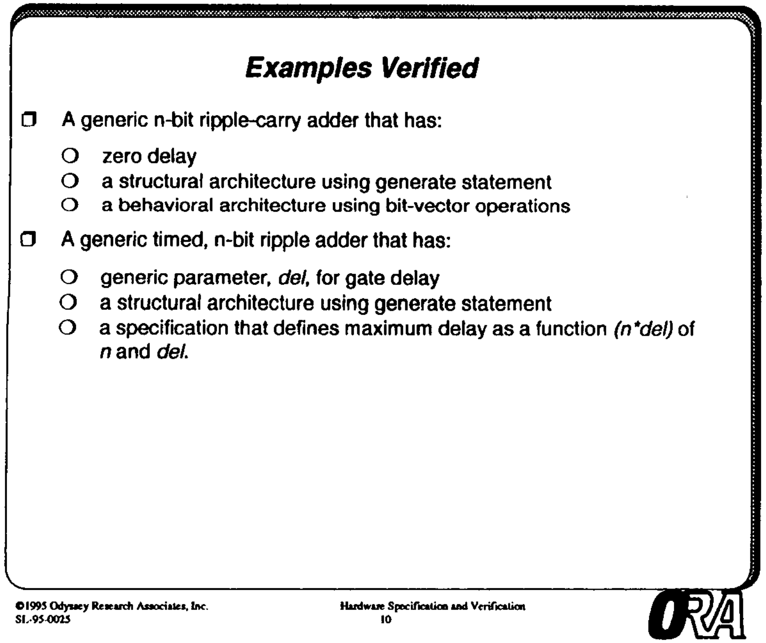
- ❒ Two-tiered approach to system specification
  - ○ Mathematics
    - ▮ Expresses the basic model of the system
    - ▮ Allows analysis of system features independent of system implementation
  - ○ Interface to Designs
    - ▮ Specifies the relationship between the actual design and the mathematical model
    - ▮ Allows analysis of the actual system implementation
- ❒ A theorem prover allows formal vèrification to be done

## System Architecture



## VHDL language constructs supported so far

- ❒ All phase I VHDL constructs are still supported
  - ○ Concurrent signal assignments
  - ○ Structural architectures
- ❒ Array types and enumeration types are supported.
- ❒ Constrained subtypes are supported.
- ❒ The declarations in package standard are supported.
- ❒ Generate statements are supported.
- ❒ Concurrent signal assignments with (transport) delay are supported.

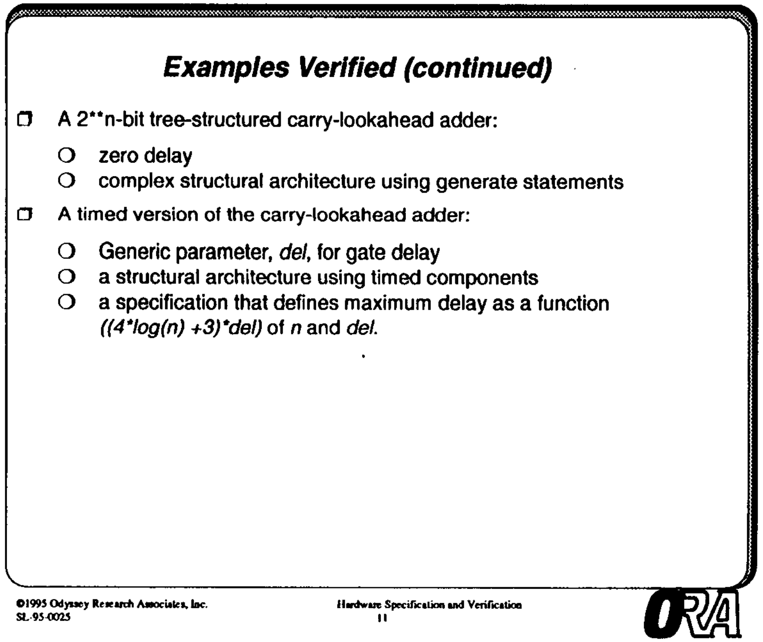


## Examples Verified

- □ A generic n-bit ripple-carry adder that has:
  - ○ zero delay
  - ○ a structural architecture using generate statement
  - ○ a behavioral architecture using bit-vector operations
- □ A generic timed, n-bit ripple adder that has:
  - ○ generic parameter, *del*, for gate delay
  - ○ a structural architecture using generate statement
  - ○ a specification that defines maximum delay as a function *(n*del)* of *n* and *del.*

©1995 Odyssey Research Associates, Inc.
SL-95-0025

Hardware Specification and Verification
10

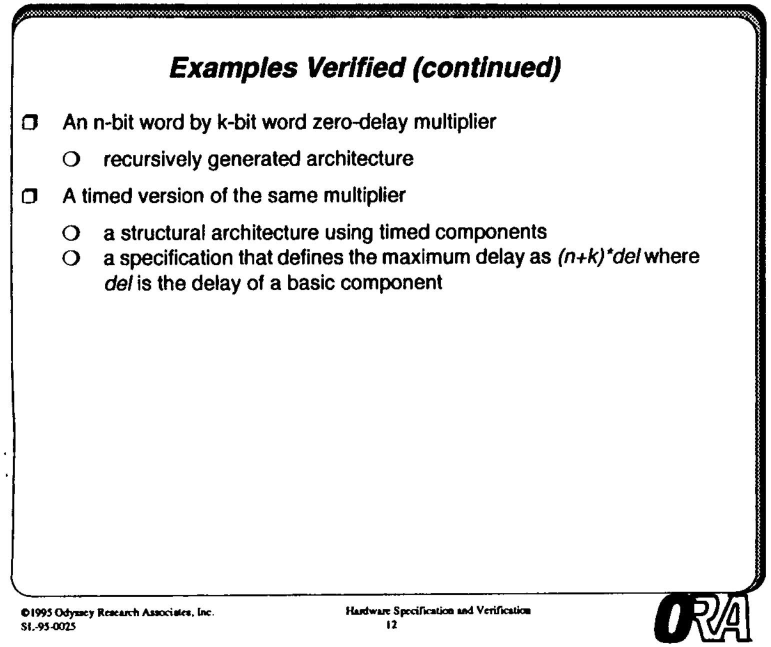


## Examples Verified (continued)

- □ A 2**n-bit tree-structured carry-lookahead adder:
  - ○ zero delay
  - ○ complex structural architecture using generate statements
- □ A timed version of the carry-lookahead adder:
  - ○ Generic parameter, *del*, for gate delay
  - ○ a structural architecture using timed components
  - ○ a specification that defines maximum delay as a function *((4*log(n) +3)*del)* of *n* and *del.*

©1995 Odyssey Research Associates, Inc.
SL-95-0025

Hardware Specification and Verification
11

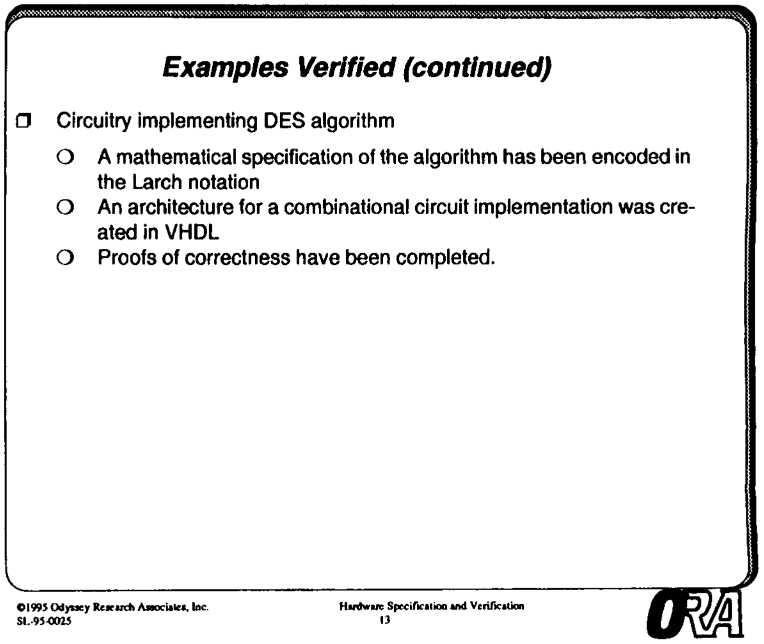


## Examples Verified (continued)

- □ An n-bit word by k-bit word zero-delay multiplier
  - ○ recursively generated architecture
- □ A timed version of the same multiplier
  - ○ a structural architecture using timed components
  - ○ a specification that defines the maximum delay as *(n+k)*del* where *del* is the delay of a basic component

©1995 Odyssey Research Associates, Inc.
SL-95-0025

Hardware Specification and Verification
12



## Examples Verified (continued)

- □ Circuitry implementing DES algorithm
  - ○ A mathematical specification of the algorithm has been encoded in the Larch notation
  - ○ An architecture for a combinational circuit implementation was created in VHDL
  - ○ Proofs of correctness have been completed.

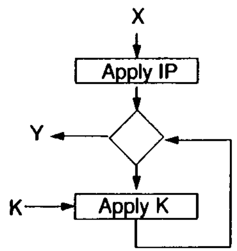
©1995 Odyssey Research Associates, Inc.
SL-95-0025

Hardware Specification and Verification
13

## DES Specification

- ❒ The DES standard consists of
  - ○ a collection of tables
  - ○ procedures for applying these tables to input words

```
K=
23 34 56 10  2 23 56 ...
34  9  8  11 3  12 12 ...
45 64 1 5 7 10 23 43 ...
...

A=e23 e34 e56 e76 ...
```



- ❒ A LARCH specification records both of these in a machine readable form suitable for various kinds of analysis

## DES Specification in Larch (Components)

```
0.1   Des
Des : trait
 includes (IP_def, K_def, E_def, S_def, P_def, Kluge)
 Operation enumeration of encode, decode
   introduces
     K : Int, Bit_vector, Operation → Bit_vector
     L0, R0 : Bit_vector → Bit_vector
     E : Bit_vector → Bit_vector
     A : Bit_vector, Bit_vector → Bit_vector
     B : Bit_vector → Bit_vector
     P : Bit_vector → Bit_vector
     R : Bit_vector, Bit_vector → Bit_vector
     Ri : Bit_vector, Bit_vector, Operation, Int, Bit_vector → Bit_vector
     Li : Bit_vector → Bit_vector
     Liter, Riter : Bit_vector, Bit_vector, Int, Operation, Bit_vector → Bit_vector
     Y : Bit_vector, Bit_vector → Bit_vector
     encode, decode : Bit_vector, Bit_vector → Bit_vector
```

## Specification of Tables (E transformation)

```
0.1   E_def
E_def : trait
 includes (Selection)
   introduces
     E : → Sel
     E : Bit_vector → Bit_vector
   asserts
     equations
     Econst : E = [32, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9, 8, 9, 10, 11, 12, 13, 12, 13, 14, 15, 16, 17, 16, 17,
                  18, 19, 20, 21, 20, 21, 22, 23, 24, 25, 24, 25, 26, 27, 28, 29, 28, 29, 30, 31, 32, 1]
     ∀ b:Bit_vector
        E : E(b) = apply(E, b)
```

## Hardware Design Implementing the DES

- ❒ A VHDL fragment is annotated with a specification

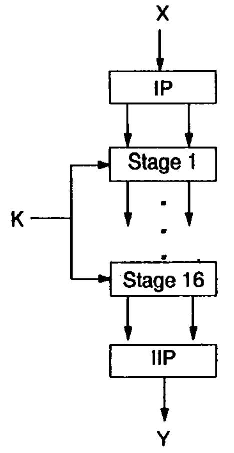
```
use work.DesTypes.all;
entity DEScipher is
  port(x : in Bit64; key : in Bit64; op : in Bit;
       y:out Bit64);
end cipher;
```

entity DEScipher
*include DES*
guarantees *if op='0' then y = encode(x,key)*
                    *else y = decode(x,key)*
*end*

## Different Architectures, Same Function

- □ Several architectures can be designed to accomplish the same function
- □ All have the same specification and their functionality is understood via the specification not the implementation
- □ The simplest architecture is a combinational circuit design that could possibly be generated directly from the specification

## Architecture 1 (Combinational)



## Architecture 1 (Structure)

```
architecture structural of cipher is
  signal l : Bit32array;
  signal r : Bit32array;
begin
  IP : ip_unit port map(x,l(0),r(0));

  LR : for i in 1 to 16 generate
    LRU : lr_unit
            generic map(i)
            port map(l(i-1),r(i-1),key,op,l(i),r(i));
    end generate;

  IIP : iip_unit port map(l(16),r(16),y);

end structural;
```

## Architecture 1 (Stage I)

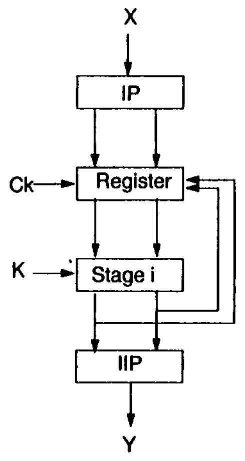
```
architecture structural of lr_unit is
  signal e,ke,kd,a : Bit48;
  signal b,p: Bit32;
begin
  E_unit : eunit     port map(ri,e);
  K_unit_e : kunit     generic map(i)       port map(key,ke);
  K_unit_d : kunit     generic map(17-i)     port map(key,kd);
  with op select
   a <= e xor ke when '0',
           e xor kd when '1';
  B_unit : bunit     port map(a, b);
  P_unit : punit     port map(b, p);
  ro <= p xor li;
  lo <= ri;
end structural;
```

## Architecture 1 (Behavior)

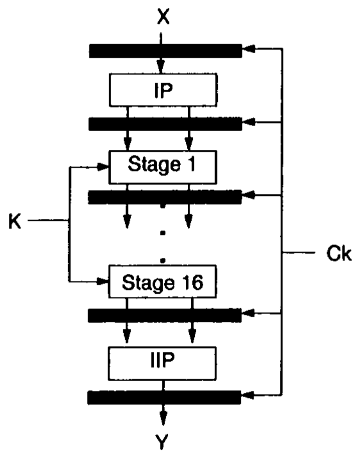
```
architecture behavior of kunit is
begin
 with i select
 k <=
key(22)&key(28)&key(39)&key(54)&key(37)&key(4)&key(47)&key(30)&key(5)&k
ey(53)&key(23)&key(29)&key(61)&key(21)&key(38)&key(63)&key(15)&key(20)&
key(45)&key(14)&key(13)&key(62)&key(55)&key(31)&key(10)&key(51)&key(34)
&key(60)&key(49)&key(17)&key(33)&key(57)&key(2)&key(9)&key(19)&key(42)&
key(3)&key(35)&key(26)&key(25)&key(44)&key(58)&key(59)&key(
1)&key(36)&key(27)&key(18)&key(41) when 1,

key(14)&key(20)&key(31)&key(46)&key(29)&key(63)&key(39)&key(22)&key(28)
&key(45)&key(15)&key(21)&key(53)&key(13)&key(30)&key(55)&key(7)&key(12)
&key(37)&key(6)&key(5)&key(54)&key(47)&key(23)&key(2)&key(43)&key(26)&k
ey(52)&key(41)&key(9)&key(25)&key(49)&key(59)&key(1)&key(11)&key(34)&ke
y(60)&key(27)&key(18)&key(17)&key(36)&key(50)&key(51)&key(58)&key(57)&k
ey(19)&key(10)&key(33) when 2
```

etc. for all 16 cases

## Architecture 2 (Clocked)



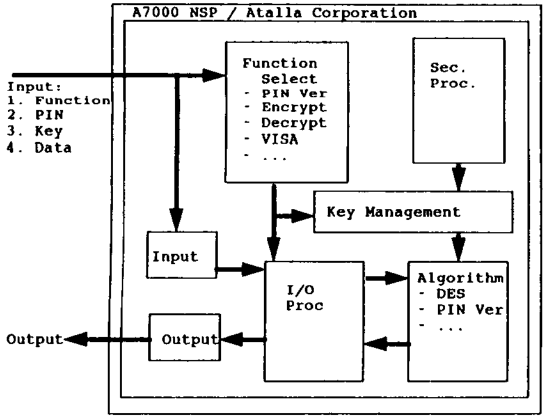
## Architecture 3 (Pipelined)



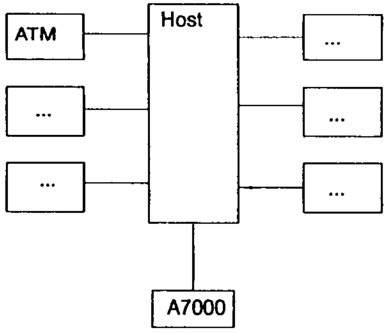
## Different Architectures, Same Function

- ❒ Architecture 1 is the simplest
- ❒ Architecture 2 saves space
- ❒ Architecture 3 is fastest
- ❒ All have the same specification and their functionality is understood via the specification not the implementation

## DES in a Larger Context



## A7000 in a larger context



## Our Methods Address

- ❒ Specification of Components
  - ○ In a notation usable by engineers
- ❒ Verification of Implementations
  - ○ In a standard design language
- ❒ Composability of Components
  - ○ and analysis of the resulting system

## Other examples to be considered

- ❒ Some examples we are currently interested in exploring
  - ○ TAXIChip Set AM7968/7969
  - ○ Intel i860 processor
  - ○ VME Bus specification

# DRS - Derivational Reasoning System

Bhaskar Bose

Derivation Systems, Inc.
5963 La Place Court, Suite 208
Carlsbad, California 92008
Tel: (619) 431-1400
bose@dvsi.com

May 11, 1995

## Abstract

The high reliability requirements for airborne systems requires fault-tolerant architectures to address failures in the presence of physical faults, and the elimination of design flaws during the specification and validation phase of the design cycle. Although much progress has been made in developing methods to address physical faults, design flaws remain a serious problem. Formal methods provides a mathematical basis for removing design flaws from digital systems.

DRS (Derivational Reasoning System) is a formal design tool based on advanced research in mathematical modeling and formal synthesis. The system implements a basic design algebra for synthesizing digital circuit descriptions from high level functional specifications. DRS incorporates an executable specification language, a set of correctness preserving transformations, verification interface, and a logic synthesis interface, making it a powerful tool for realizing hardware from abstract specifications. DRS integrates recent advances in transformational reasoning, automated theorem proving and high-level CAD synthesis systems in order to provide enhanced reliability in designs with reduced time and cost.

# DRS - Derivational Reasoning System

Bhaskar Bose

Derivation Systems, Inc.
5963 La Place Court, Suite 208
Carlsbad, CA 92008
(619) 431-1400
bose@dvsi.com

## Overview

- ◆ The Problem Domain
- ◆ Derivation and Verification
- ◆ DRS - Derivational Reasoning System
  - An Example
- ◆ Conclusions

## The Problem Domain

- ◆ Computers are being used with increasing frequency where the correct implementation is critical.
  - complex systems
  - reduced time to market
  - safety-critical applications
- ◆ Establish a rigorous framework for reasoning about complex designs and guaranteeing integrity in the design process.

## Derivation Systems, Inc.

- ◆ Corporate Mission
  - Advanced research for the development of computer engineering tools for the mathematical modeling and formal synthesis of digital hardware systems.

## Derivation and Verification: "Alternate" Modes of Formal Reasoning

- Verification
  - Constructing a post factum "proof of correctness" for a design.
- Derivation
  - Deriving a "correct by construction" design.

## Integrating Derivation and Verification

- Alternate modes of formal reasoning must be integrated if formal methods are to support the natural analytical and generative reasoning that takes place in engineering practice.

## Background

- "Synthesis of Digital Designs from Recursion Equations" (Johnson '84)
- DDD - Digital Design Derivation System (Bose, et.al. '87-94)

## Derivation Examples

- DDD-FM9001 [Bose'94]
- Scheme Machine [Burger et.al '94]
- Fault-tolerant Clock Synchronization Circuit [Miner '94]
- FM8501, FM8502 [Bose '90]
- Stop-and-Copy Garbage Collector [Boyer '89]
- SECD Machine [Wehrmeister '89]

## NASA SBIR Contract

- NASA Langley Research Center (LaRC)
- Develop a formal design tool based on advanced research in derivational reasoning, automated theorem proving, high-level synthesis, and logic synthesis.

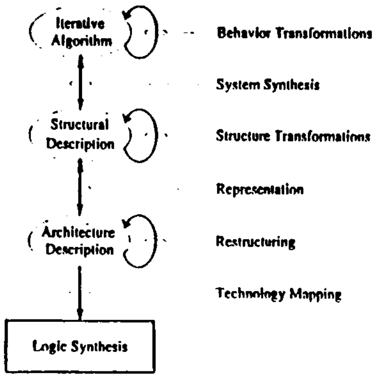
## DRS - Derivational Reasoning System

- DRS is a first-order **transformation system** which implements a basic design algebra for deriving digital circuit descriptions from high level functional specifications.
- Specifications are purely functional verifiable and executable recursive expressions.

## DRS Design Methodology

- Top-down design methodology
- Deriving a "correct by construction" design
- Correctness preserving transformations
- Ad hoc transformations
- Algebraic mechanisms for isolating verification problems to small building blocks.

## DRS Design Flow



## An Example: Fibonacci

Consider the recursive Fibonacci specification:
(The "->" denotes the conditional operator.)

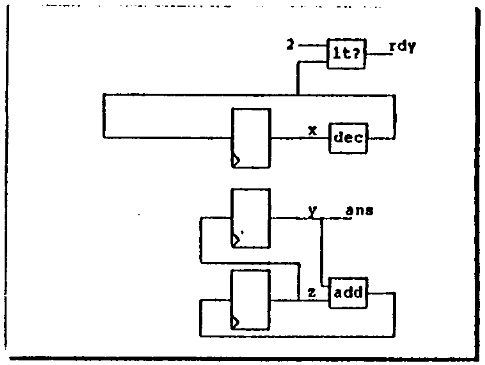
g(x, y, z) = lt?(x, 2) -> y, g(dec(x), z, add(y, z))

where

fib(x) = g(x, 1, 1)

From this recursive definition, we can derive an initial architecture:

## An Initial Architecture



## State Introduction

The first phase in the derivation is to apply a series of transformations at the behavioral level. In this phase the dec and add operations are serialized so that they may be combined into a single logic unit later in the derivation.

The first step is to introduce a function h, such that the call to g will fold within the definition of h. Introducing a definition h yields

g(x, y, z) = lt?(x, 2) -> y, g(dec(x), z, add(y, z))
h(x, y, z) = g(x, z, add(y, z))

## Folding

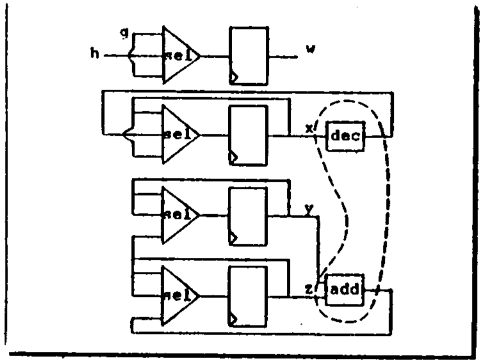
The next step is to fold the call, g(dec(x), z, add(y, z)), in g into the definition of h resulting in

g(x, y, z) = lt?(x, 2) -> y, h(dec(x), y, z)
h(x, y, z) = g(x, z, add(y, z))

This has the effect of splitting the dec and add operations between two separate definitions.

From this transformed specification, we derive the following structural description:

## The Initial Structural Specification

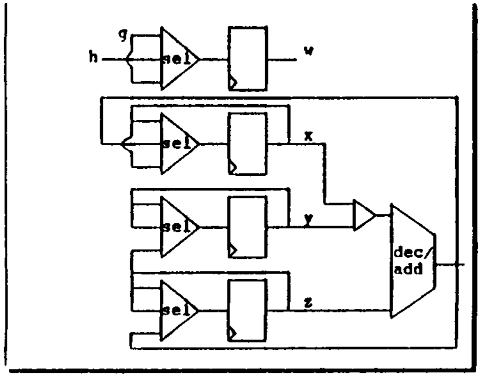


## The Initial Structural Specification (contd.)

In this phase of the derivation, transformations are applied to refine the structural specification to an architecture. The goal is to encapsulate the dec and add operations (highlighted by the dotted circle) into a single abstract component.

An algebraic transformation, called factorization, encapsulates the dec and add operations into a single component and synthesizes a behavioral description denoting the abstraction.

## Factoring: dec and add



## Further Stages of the Derivation

- ♦ Implement the abstract component with either
  - continued derivation of the abstract component
  - mapping to a verified library component
  - use of verification to replace with a particular implementation.
- ♦ Control and architecture are directly synthesized in hardware.

## Conclusion

- Derivation methodology, along with the incorporation of verification and logic synthesis, provides a powerful tool to support the natural analytical and generative reasoning that takes place in engineering practice.
- The DRS System, is a reflection of this ideal.

## Future Directions

- Continued development of DRS through Phase II SBIR Contract.
- Mechanical proofs of transformation rules.
- Integration with high-level theorem provers.
- Develop tacticals and analysis tools.
- Application of DRS to a practical design problem.

Omit

# Session 7: Researcher Perspectives on Formal Methods

**Jim Caldwell, Chair**

- David Dill, Stanford University (*No slides available*)
- Doug Hoover, Odyssey Research Associates
- Steve Johnson, Indiana University
- Natarajan Shankar, SRI International (*No slides available*)

# Formal Methods — Panel Discussion

D. N. Hoover
Odyssey Research Associates, Inc.
301 Dates Dr.
Ithaca NY 14850-1326
Internet: hoove@oracorp.com

May 24, 1995

## FM Challenges—Object Orientation

Object Orientation is a complex collection of concepts, some of which are already part of FM (come from FM), some of which have yet to be incorporated in FM.

**Static Aspects of Object Orientation**

| OO | FM |
|---|---|
| class | abstract data type, logical theory |
| subclass | theory extension |
| object | state machine |

## Dynamic Aspects of OO (Function Dispatching)

E.g. constructed type definitions can be given in pieces.

"Normal" code

```
BOOL_EXP ::= Vble V | Not BOOL_EXP | And BOOL_EXP BOOL_EXP


value assn (Vble v)  = assn v
         | (Not e)  = not (value assn e)
         | (And e1 e2) = (value assn e1) and (value assn e2)
```

Object-oriented code

```
Class Type BOOL_EXP with Function value

BOOL_EXP Clause
{
  Vble V
  value assn (Vble v) = assn v
}

BOOL_EXP Clause
{
  Not BOOL_EXP
  value assn (Not e) = not (value assn e)
}
```

etc.

## FM Support for Object Orientation

- I don't know of any FM Spec or Proof tool that supports the "decentralized" OO definition.
- Perhaps new ideas in logic are needed to properly support reasoning about dynamic subtyping.
- There are some related problems regarding formal development of concurrent programs.
- What is the right meaning of "compositional?"

## FM and Program Methodology

### Structural/Functional Specification/Design

- Less "Black box" nature than FM.
- Not entirely formal.
- Nevertheless a good start for an FM application (e.g. Statemate).
- It also includes heuristics for trying to figure out what the spec should be.

### OO Specification/Design

- In practise (e.g. Booch), seems to say a lot about how modules are related, not much about data representations, or what the modules actually do.
- Is this a problem or an opportunity?

## Assorted Theses

1. FM is very powerful and should be used for more things than telling people their specs/code are wrong.
   - Generate code, tests, docs.
2. For example, HOL or PVS as an ML code generator.
3. FM tools should be usually automatic and should give helpful information about errors as possible.
4. Proving in general is hard, but in most applications it is not hard, unless induction is involved.
5. General purpose spec languages and theorem provers are not for everybody.

   But they are *very* useful for developing specialized tools.
6. In many settings, very simple concepts and tools are helpful.
7. The idea that you can establish something by proving it instead of testing all possible cases is inconceivable to most programmers/engineers.
8. Most programmers have no idea of what has to be in a useful spec. Learning this is probably the most valuable product of FM experience.
9. God bless the FAA.
   - Better "informal" specs to start from.

- Requirements for critical software reasonably well-defined.
- There are a number of reasonable places for FM to contribute.
- Ok to start by automating existing processes, later try to replace them.

10. Tool qualification is the main obstacle to use of FM in critical systems.
11. If you use FM, you can program fancier and still be safe.

# Steve Johnson's Panel Slides

## Issues raised in formal methods

- (Butler) *"Formal methods is the application of mathematical logic to digital system design and verification"*
- *3rd generation research requires (1) expertise in the problem domain, (2) expertise in formal methods, (3) positive collaboration.*
- (Miller) *"We're not interested in doing formal methods; we're interested in verifying our systems"*
- (everybody) *How do you quantify added value?*
- (French) *" It would be nice to have some upgrade of technique rather than just an upgrade of process."*
- *Computer programmers are not engineers."*

1

## Issues raised in formal methods

- *Formal methods is a way you do your job."*
- (Miller) *"Formal methods lacks methods."*
- (Kelly) *We didn't need to train the entire project.*
- (Rushby) *"Not everybody needs to do this stuff."*

2

## Certification in formal methods

- Strong CS background
- Exposure to (hybrid) system engineering
- 3–5 courses in logic and program verification (plus math)
- Technical training in a range of tools (V&V, CASE, and CAD)
- For qualification: *experience.*

3

## Status of formal methods

○ Low critical mass in Academe

○ Application niche is still narrow

○ Can't tell when you've suceeded
(THERAC25 spec: "Don't kill anybody.")

○ Expectations are becoming more realistic.

○ Research not ready for prime time.

○ Technical transfer at scale doesn't make sense.

○ Real progress in research

○ Is horizon receding faster?

4

## Next 5–10 years

- Focus on reusable technology (algorithms, theories, clock sync, floating point)
- Develop verification/consulting industry
- Consumers *must* make demand known, and reward FM skills.
- Industry must assume some of the cost (short and long term)

## Speculation on research

- FM is like CAD was 25 years ago, but without the $ to integrate.
- Integration is *very* challenging for both engineering and academic reasons.
- General recognition of the need for a diverse tool set is leading to preliminary experience "in the lab."
- There will continue to be a gap (gulch) between research focus and applied interest.

# Session 8: Research Issues (1)

## C. Michael Holloway, Chair

- **Safety Analysis,** by *John C. Knight,* University of Virginia
- **Non-standard Analysis and Embedded Software,** by *Richard Platek,* Odyssey Research Associates

# Safety Analysis

John C. Knight
Department of Computer Science, University of Virginia
Charlottesville, VA 22903

## Case Studies

We are engaged in a research program in safety-critical computing that is based on two case studies. We use these case studies to provide application-specific details of the various research issues, and as targets for evaluation of research ideas.

The first case study is the *Magnetic Stereotaxis System* (MSS), an investigational device for performing human neurosurgery being developed in a joint effort between the Department of Physics at the University of Virginia and the Department of Neurosurgery at the University of Iowa.

The system operates by manipulating a small permanent magnet (known as a "seed") within the brain using an externally applied magnetic field. By varying the magnitude and gradient of the external magnetic field, the seed can be moved along a non-linear path and positioned at a site requiring therapy, e.g., a tumor. The magnetic field required for movement through brain tissue is extremely high, and is generated by a set of six superconducting magnets located in a housing surrounding the patient's head. The system uses two X-ray cameras positioned at right angles to detect in real time the locations of the seed and of X-ray opaque markers affixed to the patient's skull. The X-ray images are used to locate the objects of interest in a canonical frame of reference.

The second case study is the *University of Virginia Research Nuclear Reactor* (UVAR). It is a 2 MW thermal, concrete-walled pool reactor. The system operates using 20 to 25 plate-type fuel assemblies placed on a rectangular grid plate. There are three scramable safety rods, and one non-scramable regulating rod that can be put in automatic mode. It was originally constructed in 1959 as a 1 MW system, and it was upgraded to 2 MW in 1973. Though only a research reactor rather than a power reactor, the issues raised are significant and can be related to the problems faced by full-scale reactor systems.

## Safety Kernel

The software in systems like those in our case studies is very large and complex. We assume that, because of this size and complexity, faults will remain in the software for an application after development. An approach we are pursuing to deal with this is a software architecture termed a *safety kernel*, a concept directly analogous to the security kernel used in security applications.

A *security kernel* provides assurance that a set of *security policies* is enforced independently of the application program. Verification of the security kernel is sufficient to ensure enforcement of those policies encapsulated within the security kernel. The application program need not enforce the security policies, and it can, in fact, undertake actions that would normally lead to violation of the security policies with no danger of actual violations taking place. The similarity between security concerns and safety concerns is considerable and the concept of a safety kernel is appealing. If the concept were feasible, a safety kernel would enforce a set of *safety policies* by monitoring requests to devices, device actions, device status, application software status, and so on.

We have developed an enforcement safety kernel and integrated it into our MSS implementation. The safety kernel is generated automatically from a formal specification of the safety policies, and tests of the MSS instantiation show excellent performance.

## Testing

Systems of this complexity pose significant challenges in the area of testing, especially in the large number of possible test cases. We are using a technique that we call *specification limitation* to permit demonstration of useful properties by exhaustive testing. By specification limitation we mean that the specification for the application is deliberately limited in several areas to restrict the total number of test cases. For example, in the MSS the angles entered by the operator for the required direction of motion are rounded to 1/10 of a degree. In practice, this is not a significant functional restriction but it permits exhaustive testing of the angles used for setting direction. The same approach is used with distance.

A second significant problem in testing complex systems is correctness determination, i.e., determining whether the outputs are correct. In our MSS implementation, we have addressed this problem by the use of *reversal checks* on the entire system. A reversal check computes a program's input from its output and compares this with the actual input. The current calculations for the superconducting coils, for example, begin with a required force and are very complex. Computing the force resulting from the coil currents, however, is simple and provides the exact inverse of the current calculations. Thus the input can be computed and compared. A variation on the idea of a reversal check is also used by the MSS imaging subsystem.

# SAFETY ANALYSIS*

## OR
## SOME THINGS WE HAVE LEARNED FROM
## A COUPLE OF CASE STUDIES

John C. Knight

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

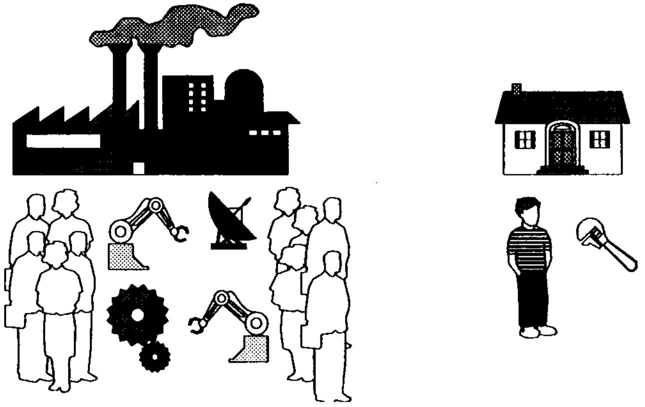
* Work sponsored in part by NASA, the NSF, the NRC, & Motorola Inc

NASA Formal Methods Workshop - Slide 1 (© John C. Knight 1995)

UVA
*Department of Computer Science*

## EXPERIENCE WITH SAFETY-CRITICAL SYSTEMS

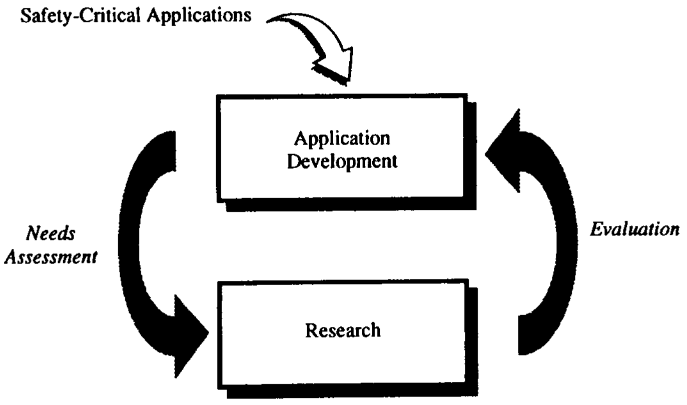
**Industry/Government** **University**



NASA Formal Methods Workshop - Slide 2 (© John C. Knight 1995)

UVA
*Department of Computer Science*

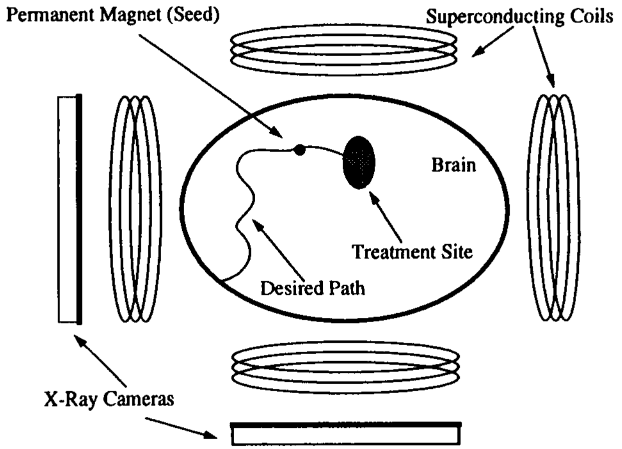
## RESEARCH BASED ON CASE STUDIES



NASA Formal Methods Workshop - Slide 3 (© John C. Knight 1995)

UVA
*Department of Computer Science*

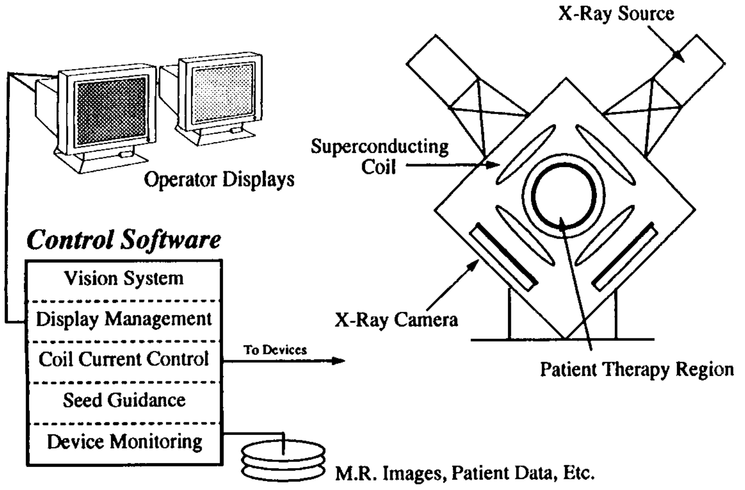
## CASE STUDY MAGNETIC STEREOTAXIS SYSTEM - CONCEPT



NASA Formal Methods Workshop - Slide 4 (© John C. Knight 1995)

UVA
*Department of Computer Science*

## Case Study - Magnetic Stereotaxis System


X-Ray Source

Operator Displays

Superconducting Coil

***Control Software***

Vision System

Display Management

Coil Current Control

Seed Guidance

Device Monitoring

To Devices

X-Ray Camera

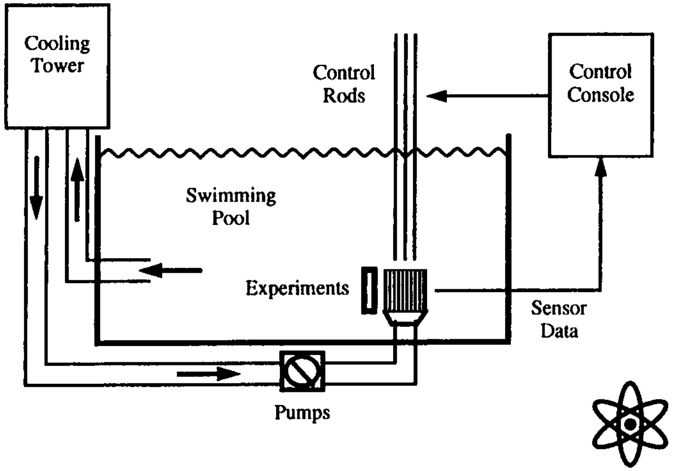
Patient Therapy Region

M.R. Images, Patient Data, Etc.

## Some Of The MSS Safety Issues

- External Superconducting Coils:
  - One Or More Coils Fail And Distorted Force Applied To Seed
  - Signals Scrambled Between Computer And Coil Controller(s)
- X-Ray Source Fails "On" When Supposed To Be "Off" Or Vice Versa
- X-Ray Camera Delivers Ghost On Image Or Incorrect Image
- Display System:
  - Wrong Seed Location Shown On Magnetic-Resonance Image
  - Wrong Magnetic-Resonance Image Displayed
- Software System:
  - Erroneous Calibration
  - Incorrect Coil Currents Calculated And Applied
  - Software Commands Any Device "On" Or "Off" Incorrectly
  - Vision System Errors - Skull Marker Or Shadow Observed Rather Than Seed

## Case Study - UVa Research Nuclear Reactor


Cooling Tower

Control Rods

Control Console

Swimming Pool

Experiments

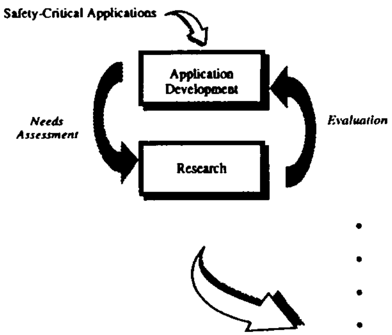
Sensor Data

Pumps

## Some Of The Reactor Safety Issues

- Uncontrolled Withdrawal Of Reactor Control Rods
- Loss Of Water In The Reactor Pool
- Coolant Pump Failure
- High Radiation Levels Outside Of The Reactor Pool
- Undocumented Alteration Of The Core Configuration
- Neutron Flux Sensor Failure
- Air Bubbles Or Other Obstructions To Cooling Inside The Core
- Incompatibility Of Instrumentation And Power Range
- High Power Operation Without Primary Cooling System

## Research Topics

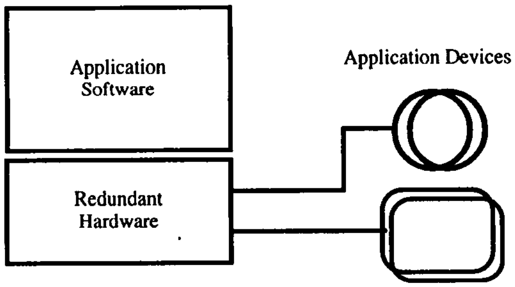


- Safety Specification Capture
- User-interface Specification
- Safety Policy Enforcement
- Software Testing
- Software Reuse
- Software Documentation
- Etc.....

UVA
*Department of Computer Science*

NASA Formal Methods Workshop - Slide 9 (© John C. Knight 1995)

## Naive Application Architecture

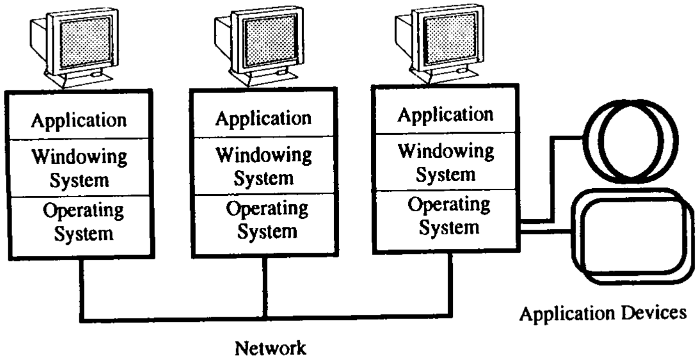


- Keep It Simple, S*****

UVA
*Department of Computer Science*

NASA Formal Methods Workshop - Slide 10 (© John C. Knight 1995)

## Realistic Application Architecture



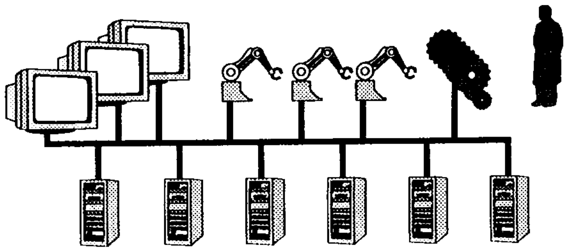
- Diverse Hardware, Network, High-performance Displays
- Extensive, Diverse And Unreliable Software, Perhaps Off-the-shelf

UVA
*Department of Computer Science*

NASA Formal Methods Workshop - Slide 11 (© John C. Knight 1995)

## The Problem We Face

- Software Is *Large And Complex* In Many Safety-critical Systems:



- Huge Subsystems, E.g. System Services, Windowing, The Application, Etc.
- How Do We Build Safety-critical Software That Is:
  - Dependable?
  - Cost-effective?

UVA
*Department of Computer Science*

NASA Formal Methods Workshop - Slide 12 (© John C. Knight 1995)

## Specification Capture

**Clearly, To Build A System, We Need Some Form Of Formal Specification.**

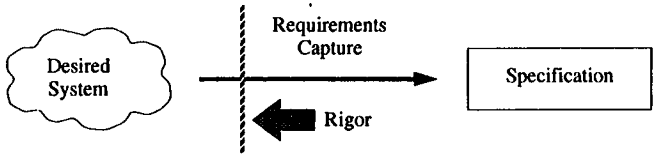
**We Chose Z** **So We Are Done** **Right?**

- Approach We (System & Software Engr's) Followed Originally With The MSS:
  - Build The System Fault Trees
  - Develop Formal Safety Specification (In Z)
- The Result Was:
  - An Informal, Ad Hoc Capture Process
  - Continual Revisions To Both The Fault Trees And The Specification
  - Very Little Confidence In The Specification
  - Confusion Over Introducing Software Concerns Into The Fault Trees
  - Confusion Between Software Coping With A Failure And Software Failing
  - Specification That Was Hard To Work With
  - Unsatisfactory Situation...

*UVA*
*Department of Computer Science*

## Requirements Capture - Process Goals

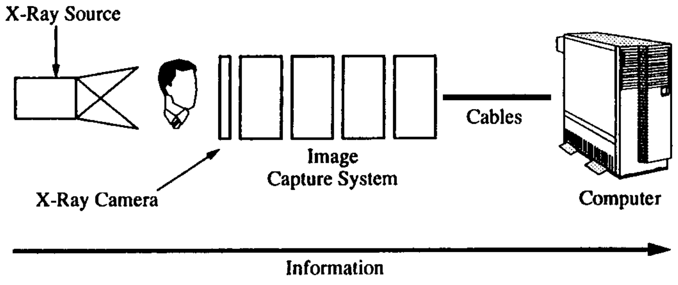
- Separation Of Software Safety And Specification Safety
- Separation Of Software Concerns And Systems Engineering Concerns
- A Process That:
  - Is Rigorous, Repeatable, And Dependable
  - Addresses Real-World Problems
  - Is Accessible To Practitioners
  - Applies To A Wide Spectrum Of Systems
  - Applies At Many Levels Of Formality



*UVA*
*Department of Computer Science*

## Information-Flow Modelling

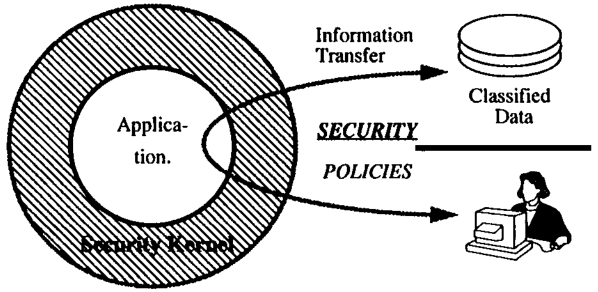
- Models Flow Of Information Through "Information Subsystem"
- Systematically Considers:
  - All Points Where Component (Including Software) Failure Can Corrupt Info.
  - All Points Where Such Failure Can Be Detected & Dealt With



*UVA*
*Department of Computer Science*

## *SECURITY* Kernel Concept

- Concept Is That Security Kernel Controls Access To All Information
- Kernel Enforces A Set Of *Security* Policies Irrespective Of Application Software's Actions:

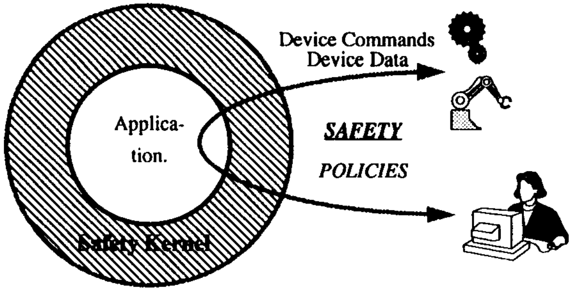


- Might A Similar Approach Work For Safety (Rushby, 1989)?

*UVA*
*Department of Computer Science*

## *SAFETY* KERNEL CONCEPT

- Concept Is That Safety Kernel Controls Access To All Devices
- Kernel Enforces Set Of *Safety* Policies Irrespective Of Application Software's Actions:



Applica-tion.

Safety Kernel

Device Commands
Device Data

***SAFETY***

*POLICIES*

- A Similar Approach Appears To Work For Safety

NASA Formal Methods Workshop - Slide 17 (© John C. Knight 1995)

*UVA*
*Department of Computer Science*

## SOME OF THE MSS SAFETY POLICIES

*If the seed moves faster than 2.0 mm/sec., the coil currents must be set to zero.*

*The coil currents must be within 5.0 amps of the predicted value.*

*The coil current requested by the application must be within the range -100 amps to 100 amps.*

*An X-ray source must be "off" for 0.2 seconds before an "on" command is executed.*

*The total X-ray dose during an operation must be less than 100 millirem.*

*Before moving the seed, a reversal check must be executed on the requested currents to compare the predicted force with the desired force.*

*And so on.....*

NASA Formal Methods Workshop - Slide 18 (© John C. Knight 1995)

*UVA*
*Department of Computer Science*

## SOME OF THE REACTOR SAFETY POLICIES

*The control rods must not be withdrawn at a rate faster than 1.5 mm/sec.*

*The position of the regulating control rod must be adjusted at least once per second based on the power of the reactor.*

*The control rods must be scrammed if a safety channel indicates a power level greater than 125% of the authorized maximum.*

*The control rods must be scrammed if the pool water level falls below 19' 3.25".*

*The control rods must be scrammed if the inlet water temperature exceeds 105° F*

*And so on.....*

NASA Formal Methods Workshop - Slide 19 (© John C. Knight 1995)

*UVA*
*Department of Computer Science*

## SUPPORT VS. ENFORCEMENT KERNELS

| ***SUPPORT*** **KERNEL:** | ***ENFORCEMENT*** **KERNEL:** |
|---|---|
| • Support provided for policy enforcement | • Enforces selected policies |
| • Application software responsible for policy enforcement | • Application software is not responsible for kernel-enforced policies |
| • Enforcement depends on correct use (or use at all) of kernel operations | • Policies enforced irrespective of use of kernel operations |
| • Support provided as potentially useful services, e.g., assertions, timers, device drivers | • Kernel not traditional - not necessarily implemented on bare hardware |
| • Examples: traditional operating systems, some earlier safety kernels | • Kernel enforces policies much like a fuse enforces a policy in an electrical system |

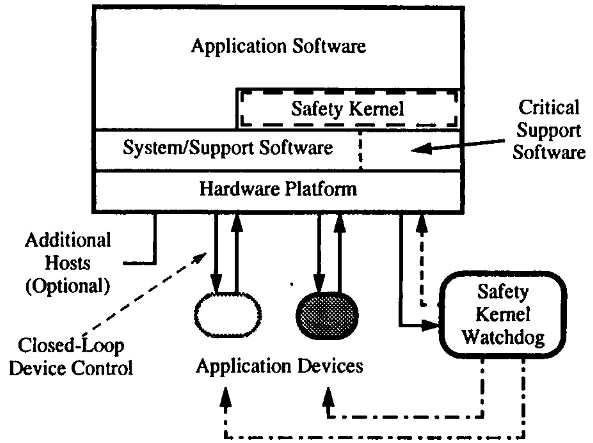
NASA Formal Methods Workshop - Slide 20 (© John C. Knight 1995)

*UVA*
*Department of Computer Science*

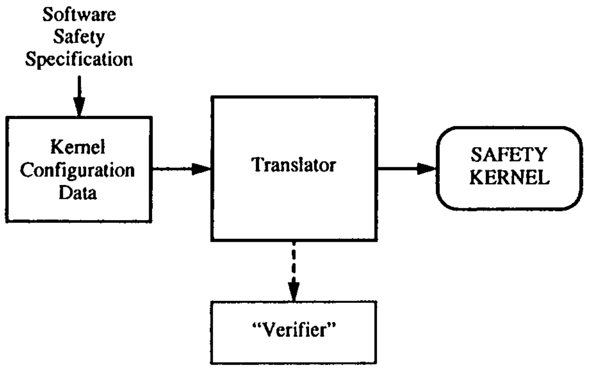
## Major Safety Kernel Issues

- How Are Policies Identified?
  - Taxonomy - Rigid Structure
  - Analysis Of Safety Specification
- What Can The Kernel Enforce?
  - Most But Not All Policies
  - Effectiveness Tradeoff
  - "Weakened" Policies
- What Is Required For Dependable Enforcement?
  - Exclusive Device Access
  - Support Services
  - Data Integrity
- How Should Kernel Be Implemented?
- How Can Kernel Be Verified?

## System Design



## Safety-Kernel Implementation



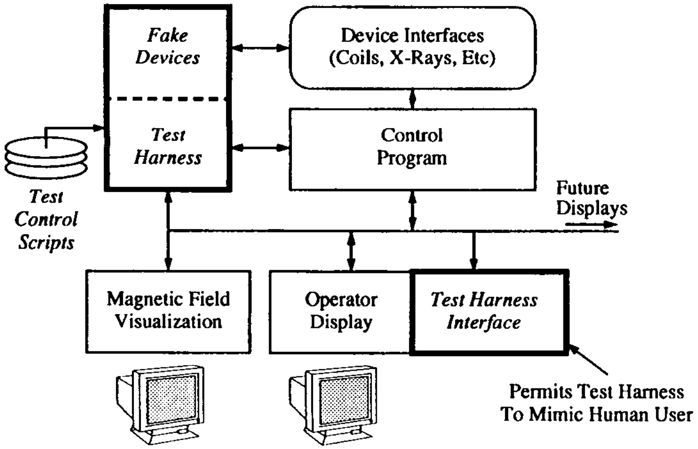
## Testing Issues

- These Systems:
  - Have Significant Graphic User Interfaces
  - Involve Complex Mathematics
  - Have Stringent Safety Requirements
- Important Research Questions:
  - How Can They Be Tested? They Are Very Complex Systems
  - What Would Testing Mean? We Would Like Meaningful Results
- Testing Safety-critical Systems:
  - Infeasible (Butler & Finelli)
  - *Worse* (Ammann, Brilliant, and Knight)

☞ ***BUT*:**

**Maybe Restricted But Useful Properties Can Be Formally Established**

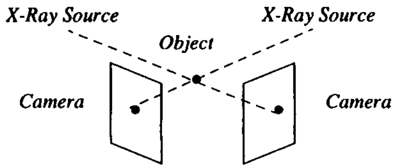
## MSS Testing Architecture



NASA Formal Methods Workshop - Slide 25 (© John C. Knight 1995)

UVA
Department of Computer Science

## MSS Test Case Selection



- Vision System Problem:

  *Two 2D X-Ray Images, Find Object In 3D Space*

- Digitized Images:

  *Number Of Object Images ~$10^9$(Ignoring Background)*

- Vision System Can Be Tested With All Possible Object Shadows:
  - Time Required < 1500 CPU Hours
  - A Useful Property Although It's Not "Correctness"

NASA Formal Methods Workshop - Slide 26 (© John C. Knight 1995)

UVA
Department of Computer Science

## Specification Limitation

☞ Deliberately Restricted The Input Space
To Improve "Testability"

- In General, Sensors Are Processed With Too Much Precision

  *E.G.: Temperature Stored As Floating Point Quantity*

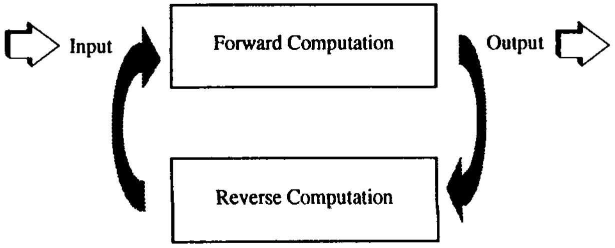
- MSS Examples:
  - User Input Seed Motion Distance Rounded To 1/10th Millimeter
  - User Input Seed Motion Angles Rounded To 1/10th Degree
- Reduces Input Space From "Huge" To ~$10^9$ Per Possible Seed Position
- Has *No* Effect On System Utility
- Permits Significant Statistical Samples If Not Exhaustive

NASA Formal Methods Workshop - Slide 27 (© John C. Knight 1995)

UVA
Department of Computer Science

## MSS Correctness Checking

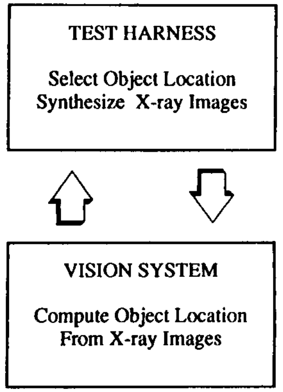
- Complicated Computations - How Do We Check Them?
- In This Case, Systematic Application Of Reversal Checks (Or Similar)
- Reversal Check:



NASA Formal Methods Workshop - Slide 28 (© John C. Knight 1995)

UVA
Department of Computer Science

## VISION SYSTEM - OBJECT LOCATION



TEST HARNESS

Select Object Location
Synthesize X-ray Images

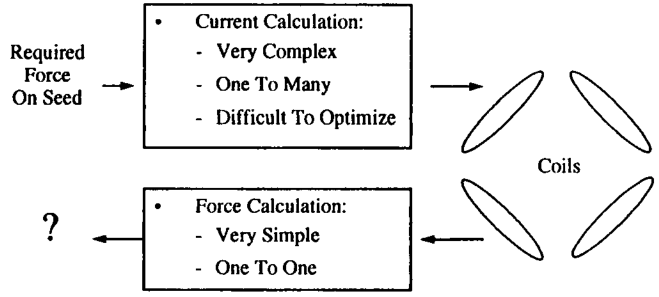
VISION SYSTEM

Compute Object Location
From X-ray Images

UVA
*Department of Computer Science*

NASA Formal Methods Workshop - Slide 29 (© John C. Knight 1995)

## MAGNETIC FIELD COMPUTATION



Required
Force
On Seed

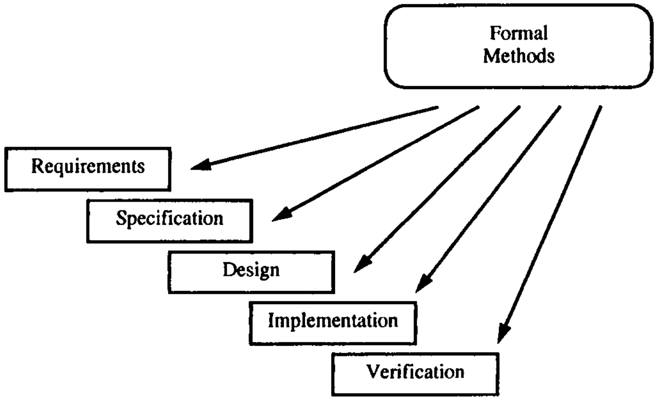
- Current Calculation:
  - Very Complex
  - One To Many
  - Difficult To Optimize

- Force Calculation:
  - Very Simple
  - One To One

UVA
*Department of Computer Science*

NASA Formal Methods Workshop - Slide 30 (© John C. Knight 1995)

## CONCLUSIONS



Formal
Methods

Requirements

Specification

Design

Implementation

Verification

UVA
*Department of Computer Science*

NASA Formal Methods Workshop - Slide 31 (© John C. Knight 1995)

# NON-STANDARD ANALYSIS AND EMBEDDED SOFTWARE

*Richard Platek*

*Odyssey Research Associates*

*richard@oracorp.com*

One model for computing in the future is ubiquitous, embedded computational devices analagous to embedded electrical motors. Many of these computers will control physical objects and processes. Such hidden computerized environments introduce new safety and correctness concerns whose treatment go beyond present Formal Methods. In particular, one has to begin to speak about Real Space software in analogy with Real Time software. By this we mean, computerized systems which have to meet requirements expressed in the real geometry of space. How to translate such requirements into ordinary software specifications and how to carry out proofs is a major challenge.

In this talk we propose a research program based on the use of non-standard analysis. Much detail remains to be carried out. The purpose of the talk is to inform the Formal Methods community that Non-Standard Analysis provides a possible avenue of attack which we believe will be fruitful.

# Non-Standard Analysis and Embedded Software

*Richard Platek*

*Odyssey Research Associates*
*301 Dates Drive*
*Ithaca, NY 14850-1326*
*richard@oracorp.com*

*3rd NASA Formal Methods Workshop*

*May 12, 1995*

## Outline of Talk

□ Embedded software

□ Need for formal methods to include continuous mathematics

□ A proposed research program based on non-standard analysis

## Embedded Software

□ Hidden ubiquitous computers – The environment of the future

□ Analogy with electric motors

  ○ Question: How many electric motors service you a day?

□ Many of these computational engines will directly interact with the physical environment (conventional use of the term "Embedded Software").

□ Basic specification and verification issues need to be expressed in terms of the physical environment.

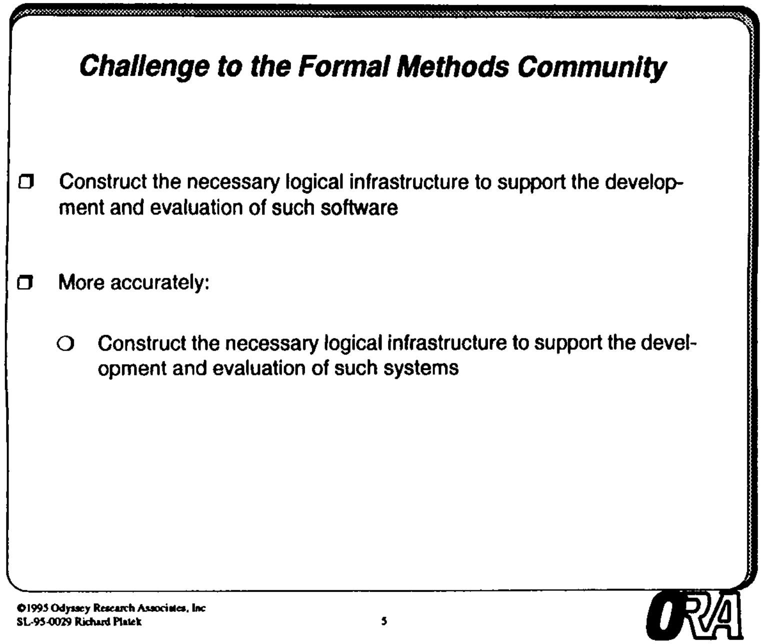
## "Real Space" Software

□ Analogy with real time software

□ Although real time software has a history going back to the earliest days of computer science it is still an active research area as to how to specify, implement and verify such software.

□ Logics for reasoning about real time software use real time (i.e., the real numbers used to measure time) or at least a close approximation to it.

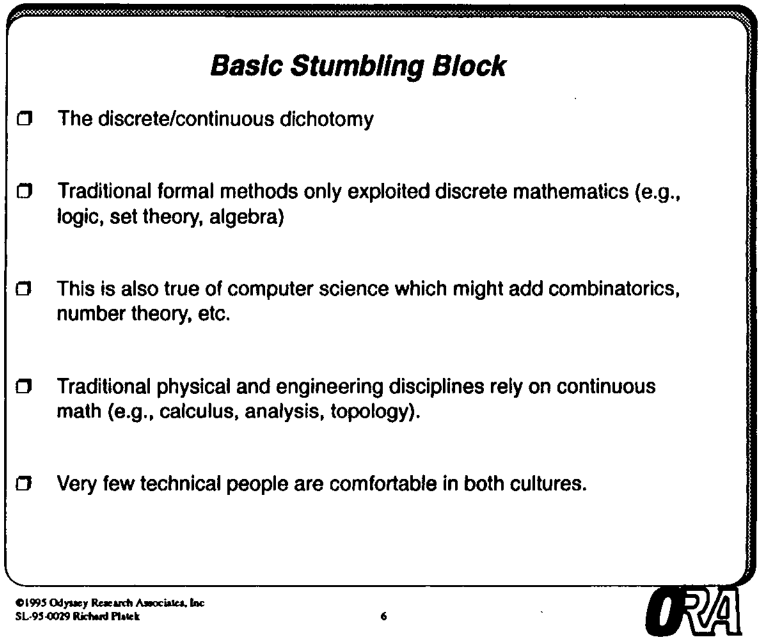
□ Real space software needs analogous logics in order to be able to prove claims such as "This software moves this object to this place at this time".



## Challenge to the Formal Methods Community

- ❒ Construct the necessary logical infrastructure to support the development and evaluation of such software
- ❒ More accurately:
  - ○ Construct the necessary logical infrastructure to support the development and evaluation of such systems

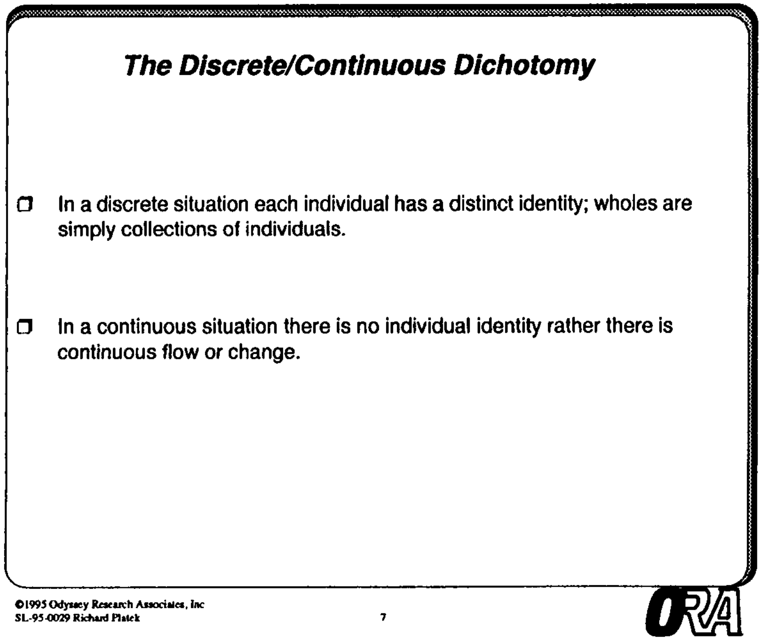
5



## Basic Stumbling Block

- ❒ The discrete/continuous dichotomy
- ❒ Traditional formal methods only exploited discrete mathematics (e.g., logic, set theory, algebra)
- ❒ This is also true of computer science which might add combinatorics, number theory, etc.
- ❒ Traditional physical and engineering disciplines rely on continuous math (e.g., calculus, analysis, topology).
- ❒ Very few technical people are comfortable in both cultures.

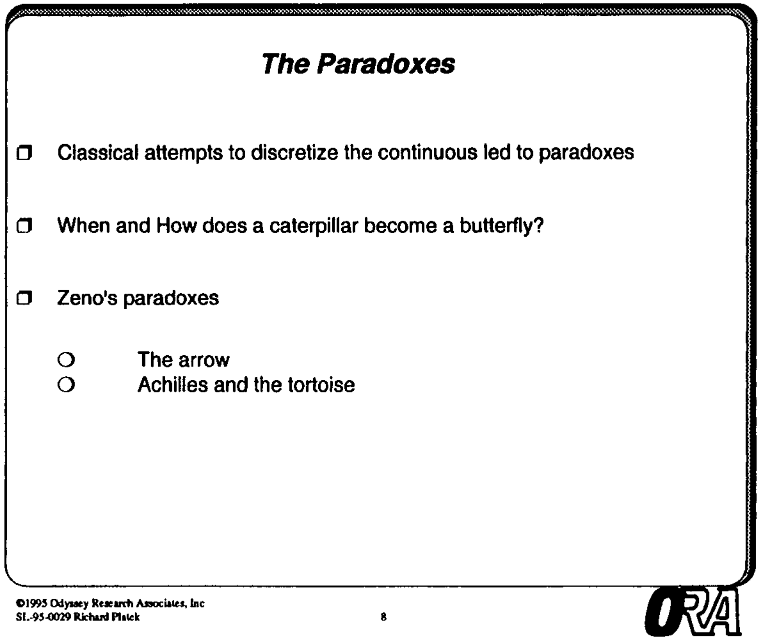
6



## The Discrete/Continuous Dichotomy

- ❒ In a discrete situation each individual has a distinct identity; wholes are simply collections of individuals.
- ❒ In a continuous situation there is no individual identity rather there is continuous flow or change.

7

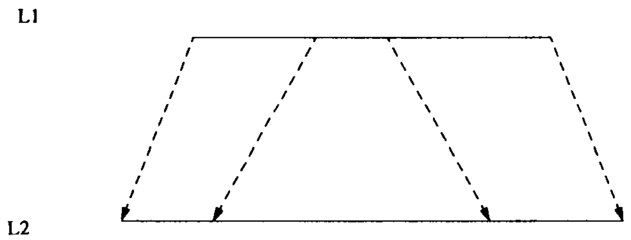


## The Paradoxes

- ❒ Classical attempts to discretize the continuous led to paradoxes
- ❒ When and How does a caterpillar become a butterfly?
- ❒ Zeno's paradoxes
  - ○ The arrow
  - ○ Achilles and the tortoise

8

## The Paradoxes



There is a one to one correspondence between the points of L1 and L2. Each point has the same length. Where do the different lengths of the lines come from?

□ Both lines have the same number of points; each point has the same length (viz., 0). Where does the length come from?

## Quotations

□ "No continuum can not be made up out of indivisibles, as for instance a line out of points, granting that the line is continuous and the point indivisible. " Aristotle

□ "A point may not be a constituent part of a line" Leibniz

□ "A true continuum has no points". Rene Thom

## Non-Standard Analysis

□ A modern, logically sound discretization of the continuous

□ Based on Leibniz's Theory of Infinitesimals

□ latter was used to develop all analysis during the 17th, 18th, and first half of the 19th century.

□ It continued to be used in applied math and engineering until mid 20th century.

□ Highly intuitive, very algebraic – there are no limit arguments.

## Leibniz' Infinitesimals

□ Positive quantities smaller than all positive reals.

□ They are not indivisible! If dx is an infinitesimal so is $\frac{dx}{2}$.

□ The extended real numbers satisfy "all" the algebraic laws of the ordinary reals.

□ Solves the paradoxes.

□ Basis for an explosive development of new mathematics.

## Examples

□ Differentiation

$$\frac{(x+dx)^2-x^2}{dx} = \frac{x^2+2\times dx+(dx)^2-x^2}{dx}$$

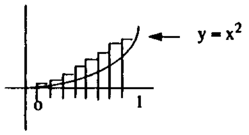
$$= 2x + dx$$

$$\approx 2x$$

No matter what choice of infintesimals $dx$

## Examples

□ Integration



## Examples (cont'd)

$$\sum_{k=1}^{n}\left(\frac{k}{n}\right)^2\cdot\frac{1}{n} = \frac{1}{n^3}\sum_{k=1}^{n}k^2$$

$$= \frac{1}{n^3}\left(\frac{n(n+1)(2n+1)}{6}\right)$$

$$= \frac{1}{n^3}\left(\frac{2n^3+3n^2+n}{6}\right)$$

$$= \frac{1}{n^3}\left(\frac{\left(n^2+n\right)(2n+1)}{6}\right)$$

$$= \frac{1}{3}+\frac{1}{2n}+\frac{1}{6n^2}$$

Let n be infinite $\approx \frac{1}{3}$ independent of n.

## Examples

□ Series

$$\cos n\theta + i \sin n\theta = (\cos\theta + i \sin\theta)^n$$

$$= \sum_{k=0}^{n} \binom{n}{k} i^k \sin^k\theta \cos^{n-k}\theta$$

$$\sum_{k=0}^{n} \frac{n!}{k!\,(n-k)!} i^k \sin^k\theta \cos^{n-k}\theta$$

## Examples (cont'd)

Given a finite angle x let n be infinite and $\theta = \frac{x}{n}$ be infinitesimal

Then

$$\frac{n!}{k!\,(n-k)!} \approx \frac{n^k}{k!}$$

$$\sin\theta \approx \theta$$

$$\sin^k\theta \approx \theta^k$$

$$\cos\theta \approx 1$$

$$\cos^{n-k}\theta \approx 1$$

$$\cos x + i \sin x \approx \sum_{k=0}^{\infty} i^k \left(\frac{x^k}{k!}\right)$$

## Applications to Embedded Software

□ Add extended reals *R and extended integers *Z as types to any given program language together with the usual arithmetic and logical operators over these types.

□ Extend the notion of computation to include possibly infinite computations to compute integrals and series.

□ Prove that a program satisfies a given spec or generate (synthesize) the program from the spec.

□ Note that computed results are not equal to the desired number they are only infinitesimally close (as in the above examples).

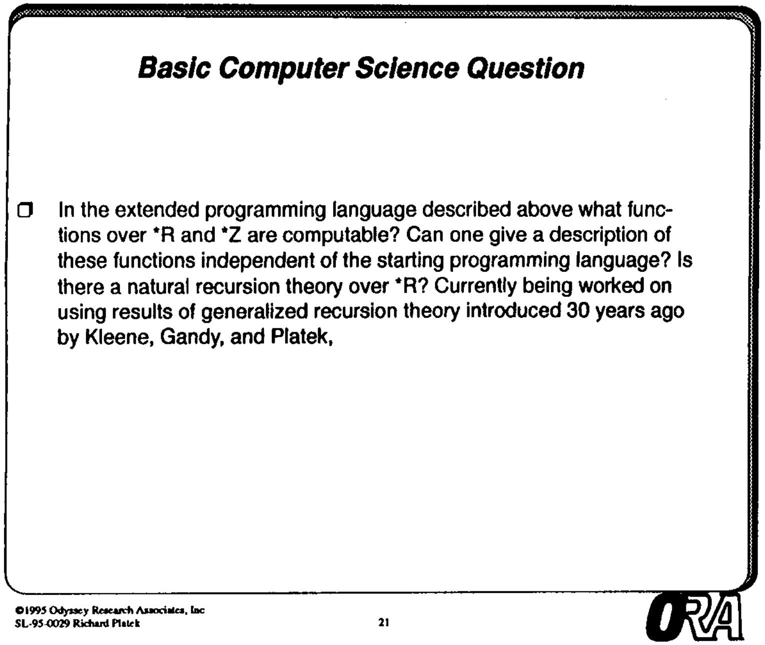
## Applications to Embedded Software (con't)

□ Analyze the problem as presented in physical terms to determine a finite epsilon such that epsilon approximate solution would be adequate.

□ Using this epsilon transform the program over *R which is infinitesmally close to the correct solution to an ordinary program which is epsilon close. How to do this is still a research topic.

□

□ The resulting program will be Real Space correct but not Real Time correct since no attention was paid to efficiency of computation. It remains to investigate how to integrate these two requirements.



## Basic Computer Science Question

□ In the extended programming language described above what functions over *R and *Z are computable? Can one give a description of these functions independent of the starting programming language? Is there a natural recursion theory over *R? Currently being worked on using results of generalized recursion theory introduced 30 years ago by Kleene, Gandy, and Platek,

omit

# Session 9: Research Issues (2)

## Victor Carreno, Chair

- **Hybrid Fault Algorithms**, by *Pat Lincoln*, SRI International
- **Model Checking**, by *David Dill*, Stanford University

# Hybrid Fault Algorithms

Patrick Lincoln, SRI International

In Association With:

J. Rushby SRI

N. Suri Allied Signal

C. Walter (then) Allied Signal

Sponsored by the NASA Langley Research Center
under contract NAS1 18969

## Application: Digital Flight Control

- Extreme Reliability ($10^{-9}$ system failures/hour)
- Synchronous Systems
- NASA RCP

## Byzantine Fault Tolerant Machine Designs

- SRI's SIFT (early 80's) was the first; used 70% CPU cycles on overhead for voting etc.
- Allied-Signal's MAFT (late 80's) used hardware assistance; numerous innovations
- Draper Lab's Asymmetric FTP: less hardware
- Others: TU Vienna MARS, AIPS, FTTP...

**The Algorithms Used are Complex
Single Points of Failure**

## General Points About Verification

- Informal Proofs are Unreliable:
  In the past, have Found and Corrected Flaws in Published Algorithms and Proofs.
- PVS is an Effective Tool:
  Emphasis on Minimizing Human Effort
- Proof Reuse Reduces Cost
- Still Expensive to Prove Anything Interesting

## General Points About Algorithms

- Slight Modifications to Traditional Algorithms
  - Can Provide Much Greater Reliability
  - Can Destroy Correctness
- Hybrid Approach Widely Applicable
  - Clock Synchronization
  - Byzantine Agreement
  - Diagnosis

## Byzantine Agreement

Set of $n$ Generals (Processors)

$m$ of them are Traitors (faulty)

All Good Generals Must Agree on Plan of Attack
All Good Processors Must Agree on Sensor Value

Make NO Assumptions about Behavior of Traitors
Make NO Assumptions about Behavior of Faults

Assume One Commander, Others Are Lieutenants
Assume One Transmitter, Others Are Receivers

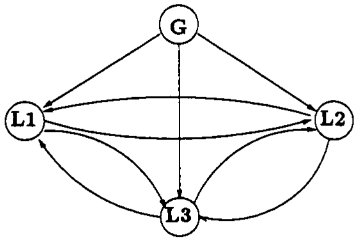
## The Byzantine Generals Problem: Requirements

Agreement: Nonfaulty Lieutenants Agree on the Value Received from the Commander

Validity: If the Commander is Nonfaulty, the Value Received by Nonfaulty Lieutenants is the Value Actually Sent

## The Oral Messages Algorithm (OM) (Lamport, Shostak, Pease 1982)

- The Commander Sends Value to Each Lieutenant
- If $r = 0$, Each Lieutenant Accepts A Value
  Otherwise, each lieutenant plays the part of the general in $OM(r-1)$ to send value received to other lieutenants
- Each Lieutenant Takes Majority Vote of the Values Received



## Properties of OM

- For Agreement and Validity Requires $n > 3a$ (Best Possible Bound)
- $r + 1$ rounds of message exchange
- Exponential Number of Messages (Moses 93)
- Need Four Channels to Withstand a Single Fault
- Formally verified (Bevier & Young '92, Rushby '92, Shankar '93)

## Hybrid Fault Models

- Extreme Positions:
  - Byzantine Approach: Components are Perfect or Failed in Unknown Manner
  - FMEA Approach: Components Fail in (many) Known Ways; Design Countermeasures for each (and combinations)
- Allied Signal's Hybrid Fault-Models:
  Include Byzantine Case, Plus a Few Common Cases (Fail-Stop, Stuck-At, $\cdots$)

## Formalizing Hybrid Fault Models: Motivation

- Maximum Assurance
- Maximum Reliability
- Minimum Hardware

## Algorithms for Hybrid Fault Models

- Extend Byzantine Fault-Tolerant Algorithms to Deal Better With NonByzantine Faults
- Survive a Few Bad Faults
  or a Lot of Simple Faults with the Same Protocol
- Allied Signal's Reliability Analysis:
  This Provides Superior Reliability
  (McElvany-Hugue '93)

## A Hybrid Fault Model

- Allied Signal (Thambidurai and Park '88)
  - ($a$) Arbitrary (Byzantine, Asymmetric)
  - ($s$) Symmetric (Value, Stuck-At)
  - ($c$) Crash (Manifest, Stopped, Out-of-Bounds)
  - ($g$) Good (Nonfaulty, OK)

## Requirements for Hybrid Agreement

Agreement: All nonfaulty receivers agree on the value ascribed to the transmitter.

Validity: If receiver $q$ is nonfaulty, the value it ascribes to the transmitter is

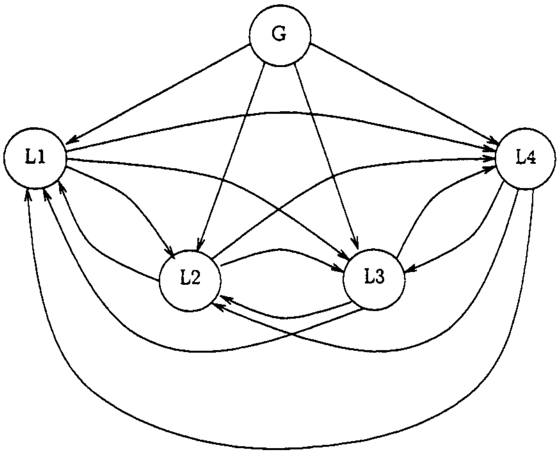
- The correct value if the transmitter is nonfaulty.
- The value actually sent to all receivers if the transmitter is symmetric-faulty.
- The distinguished value $E$ if the transmitter is manifest-faulty.

## Allied Signal's Algorithm Z (Thambidurai and Park 1988)

- Like OM
- When Manifestly Bad Value Received, Record E
- Lieutenants Pass on E When Receive E
- Ignore E in majority vote

## Algorithm Z: Flawed

- $n = 5$, $r = 1$, the Commander has a Crash Fault, one Lieutenant is a Traitor
- Good Lieutenants will believe whatever they receive from the Traitor: Protocol Fails
- MAFT Implementation Correct



## Importance of Validation

- **Lincoln & Rushby Specified Incorrect Algorithm**
- **Like Z, But Lieutenants Pass on RE "Reported Error"**
- **With Incorrect Axiom "Hybrid_Majority_Vote_1"**
  - **"Ignore Votes of Crashed Processors"**
  - **1-Round Case: OK**
  - **2-Round Case: Impossible**
  - **Should Have Been: "Ignore E Values"**
- **With Axiom, "Verified" Incorrect Algorithm**
- **Should Validate Axioms With Implementation**

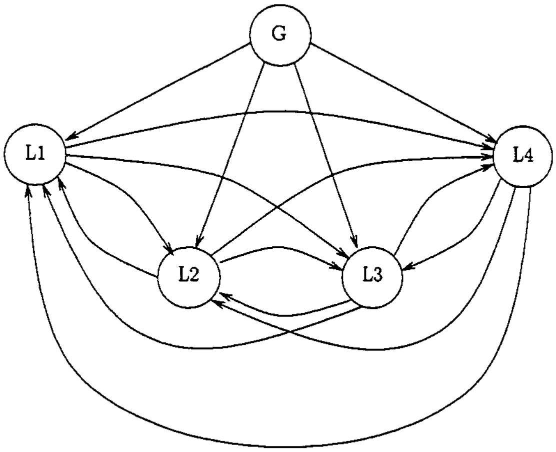
## Algorithm OMH (Lincoln & Rushby '93)

- **Like OM, but**
- **Add $m + 1$ kinds of error value: E, RE, R$^2$E etc.**
- **If receive R$^i$E, pass on as R$^{i+1}$E**
- **Ignore E in majority vote**
- **If vote yields R$^j$E, selected value is R$^{j-1}$E**
- **Formally Specified and Verified in PVS (Lincoln & Rushby '93) Validated Axioms With MJRTY Implementation**

**Doubt Anyone Could Get This Right**
**Without Formal Verification**

## Algorithm OMH

$n = 5, r = 1$



## Algorithm OMH

- **If $r \geq a$, then OMH($r$) Tolerates**

  $n > 2a + 2s + c + r$

**If $r = a$, and $s = c = 0$, Then $n > 3a$**
**(Same as Traditional Bound)**

- **($a$) Arbitrary (Byzantine, Asymmetric)**
- **($s$) Symmetric (Value, Stuck-At)**
- **($c$) Crash (Manifest, Stopped, Out-of-Bounds)**
- **($g$) Good (Nonfaulty, OK)**

## Fault Combinations Tolerated

| | Number of Faults | | |
|---|---|---|---|
| | Arbitrary (byzantine) | Symmetric (value) | Crash (manifest) |
| OM | 1 | 0 | 0 |
| | 0 | 1 | 0 |
| | 0 | 0 | 1 |

| | Number of Faults | | |
|---|---|---|---|
| | Arbitrary (byzantine) | Symmetric (value) | Crash (manifest) |
| OMH | 1 | 1 | 0 |
| | 1 | 0 | 2 |
| | 0 | 2 | 0 |
| | 0 | 1 | 2 |
| | 0 | 0 | 5 |

## Achieved Reliability: 6 Processor OMH

- **McElvany-Hugue's Reliability Analysis**
- **Reliability Goal: $10^{-9}$ System Failures/Hour**
  - **6 Processor OM: $1.5 \times 10^{-7}$**
  - **6 Processor OMH: $1.5 \times 10^{-11}$**

## Formal Specification of OMH in PVS

```
omh[m : nat, n : posnat, T : TYPE, error : T, R, UnR : [T → T]] : THEORY
BEGIN
 ASSUMING   act_ax : ASSUMPTION (∀ (t : T) : R(t) ≠ error)
            unact_ax : ASSUMPTION (∀ (t : T) : UnR(R(t)) = t)
 ENDASSUMING
rounds : TYPE  =  upto[m]
t : VAR T
fcu : TYPE  =  below[n]
fcuset : TYPE  =  setof[fcu]
fcuvector : TYPE  =  [fcu → T]
G, p, q, z : VAR fcu
v, v1, v2 : VAR fcuvector
caucus : VAR fcuset
r : VAR rounds
IMPORTING finite_cardinality[fcu, n, identity[fcu]],
          filters[fcu],
          card_set[fcu, n, identity[fcu]],
          hybrid_mjrty[T, n, error]

statuses : TYPE  =  {arbitrary, symmetric, manifest, nonfaulty}
status : [fcu → statuses]
a(z) : bool  =  arbitrary(status(z))
s(z) : bool  =  symmetric(status(z))
c(z) : bool  =  manifest(status(z))
g(z) : bool  =  nonfaulty(status(z))
as(caucus) : fcuset  =  filter(caucus, a)
ss(caucus) : fcuset  =  filter(caucus, s)
cs(caucus) : fcuset  =  filter(caucus, c)
gs(caucus) : fcuset  =  filter(caucus, g)

fincard_all : LEMMA
 |caucus|  =  |as(caucus)|  +  |ss(caucus)|  +  |cs(caucus)|  +  |gs(caucus)|

send : [T, fcu, fcu → T]
send1 : AXIOM g(p)  ⊃  send(t, p, q)  =  t
send2 : AXIOM c(p)  ⊃  send(t, p, q)  =  error
send3 : AXIOM s(p)  ⊃  send(t, p, q)  =  send(t, p, z)
send_lemma : LEMMA ¬ a(p)  ⊃  send(t, p, q)  =  send(t, p, z)

HMajority(caucus, v) : T  =  proj_1(hybrid_mjrty(caucus, v, n))

HMajority1 : LEMMA
 |gs(caucus)|  >  |as(caucus)|  +  |ss(caucus)|
    ∧  (∀ p :  g(p)  ∧  p ∈ caucus  ⊃  v(p)  =  t)
     ∧  t  ≠  error  ∧  (∀ p :  c(p)  ∧  p ∈ caucus  ⊃  v(p)  =  error)
    ⊃  HMajority(caucus, v)  =  t
```

```
HMajority2 : LEMMA
 (∀ p :  p ∈ caucus  ⊃  v1(p)  =  v2(p))
    ⊃  HMajority(caucus, v1)  =  HMajority(caucus, v2)

OMH(G, r, t, caucus) :  RECURSIVE fcuvector  =
   IF r  =  0
   THEN (λ p :  send(t, G, p))
   ELSE (λ p : IF p  =  G
               THEN send(t, G, p)
               ELSE UnR(HMajority(caucus − {G},
                                   (λ q : OMH(q, r − 1, R(send(t, G, q)), caucus − {G})(p))))
               ENDIF)
   ENDIF
  MEASURE (λ G, r, t, caucus →  nat :  r)

Validity_Prop(r) : bool  =
¬ a(q)  ∧  p ∈ caucus  ∧  q ∈ caucus
   ∧  |caucus|  >  2  ×  (|as(caucus)|  +  |ss(caucus)|)  +  |cs(caucus)|  +  r
   ⊃  OMH(q, r, t, caucus)(p)  =  send(t, q, p)

Validity :  LEMMA Validity_Prop(r)

Validity_Final : THEOREM
 g(p)  ∧  ¬ a(G)   ∧  2  ×  |a|  +  2  ×  |s|  +  |c|  +  m  <  n
   ⊃  OMH(G, r, t, fullset[fcu])(p)  =  send(t, G, p)

Agreement_Prop(r) : bool  =
 g(p)  ∧  g(q)  ∧  p ∈ caucus  ∧  q ∈ caucus  ∧  z ∈ caucus
   ∧  |caucus|  >  2  ×  (|as(caucus)|  +  |ss(caucus)|)  +  |cs(caucus)| + r
   ∧  r  ≥  |as(caucus)|
 ⊃  OMH(z, r, t, caucus)(p)  =  OMH(z, r, t, caucus)(q)

Agreement :  LEMMA Agreement_Prop(r)

Agreement_Final : THEOREM
 g(p)  ∧  g(q)  ∧  |a|  ≤  m  ∧  2  ×  |a|  +  2  ×  |s|  +  |c|  +  m  <  n
   ⊃  OMH(G, m, t, fullset[fcu])(p)  =  OMH(G, m, t, fullset[fcu])(q)
 END omh
```

## Axioms (Assumptions)

send1 : $\text{AXIOM}\ g(p)\ \supset\ \text{send}(t,p,q)\ =\ t$

send2 : $\text{AXIOM}\ c(p)\ \supset\ \text{send}(t,p,q)\ =\ \text{error}$

send3 : $\text{AXIOM}\ s(p)\ \supset\ \text{send}(t,p,q)\ =\ \text{send}(t,p,z)$

send_lemma : LEMMA
$\neg\ a(p)\ \supset\ \text{send}(t,p,q)\ =\ \text{send}(t,p,z)$

## Theorem 1

Validity_Final : THEOREM
$(g(p)\ \wedge\ \neg\ a(G)$
$\wedge\ 2\ \times\ |a|\ +\ 2\ \times\ |s|\ +\ |c|\ +\ r\ <\ n)$
$\mathit{mbox}\ \supset\ \text{OMH}(G,r,t,\text{fullset[fcu]})(p)\ =\ \text{send}(t,G,p)$

**Algorithm OMH($r$) satisfies Validity if there are more than $2(a+s)+c+r$ processors.**

**Validity: If The Commander Is not Byzantine-Faulty, The Value Received By Nonfaulty Lieutenants Is The Value Actually Sent**

## Style of Proof
## Bevier & Young, Rushby, Shankar

- **Inductive Property Definition**
- **Induction Property Assertion (Lemma)**
- **Theorem in Final Form**

Validity_Prop($r$) : bool =
$\neg\ a(q)\ \wedge\ p \in \text{caucus}\ \wedge\ q \in \text{caucus}$
$\wedge\ |\text{caucus}|\ >\ 2\ \times\ (|\text{as(caucus)}|\ +\ |\text{ss(caucus)}|)$
$+\ |\text{cs(caucus)}|\ +\ r$
$\supset\ \text{OMH}(q,r,t,\text{caucus})(p)\ =\ \text{send}(t,q,p)$

Validity : LEMMA Validity_Prop($r$)

Validity_Final : THEOREM
$(g(p)\ \wedge\ \neg\ a(G)$
$\wedge\ 2\ \times\ |a|\ +\ 2\ \times\ |s|\ +\ |c|\ +\ r\ <\ n)$
$\supset\ \text{OMH}(G,r,t,\text{fullset[fcu]})(p)\ =\ \text{send}(t,G,p)$

## While ReRunning Proof, We Noticed

**Lemma If $c=0$ (No Crash Faults), OMH $\approx$ OM**

**OM Is Better Than Earlier Claimed:**

**Corollary 1 For any $r$, Algorithm OM($r$) satisfies Validity if there are more than $2(a+s)+r$ processors.**

**Corollary 2 For any $r$, Algorithm OM($r$) satisfies Agreement if there are more than $2(a+s)+r$ processors and $r \geq a$.**

**OMH = OM + Diagnosis of Manifest Faults**

## More Corollaries

More Theorems Derived from our New Improved Understanding

**Corollary 3:** No Good Processors Are Confused By Crash Faults: If $a = s = 0$ (No Byzantine, No Symmetric Faults), OMH is optimal.

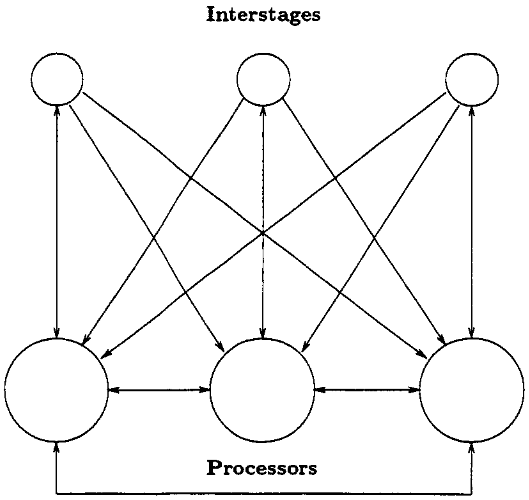
**Corollary 4:** Error Values (RE, $R^2E, \cdots$) Can Overlap With Real Values.

**Corollary 5:** (Slightly Modified) OMH($r$) Allows Implementations With Only $r + 1$ Error Values.

## FTP Architecture

- Less Total Hardware
- Only need 3 voters to mask 1 Byzantine fault (!)
- Technology used in AIPS, FTTP, etc.

## FTP Architecture



## Algorithm OM-FTP

- Lala 1986
- Similar to one-round OM with symmetric voters
- Formally specified in EHDM<br>(Harper, Alger, & Lala '91)<br>No proofs
- Formally specified in PVS<br>(Lincoln & Rushby '94)<br>Full formal proofs completed in PVS<br>Couple of days

## Hybrid Faults in FTP

- Integrate OMH and OM-FTP
- Resist more (simple) faults with less hardware

33

## Algorithm OMH-FTP (Lincoln & Rushby 1994)

- Combine OMH and OM-FTP
- Voters pass on E as RE
- Interstages pass on E as E
- Voters use value reflected from interstage, NOT original value

34

## Formal Verification in PVS

- Proof Reuse
- Copy-and-Edit Proof of OM-FTP
- Grab parts of Proof of OMH
- M-x Edit Proof
- Rerunning Edited Proofs
- Couple of Days Effort

35

## Copy-and-Edit a Proof

- Emacs Interface to Edit Proof Commands
- Kill and Yank
- Mix-And-Match Partial Proofs
- Rerun Until Proof Fails

36

## Authenticated Agreement

- Use Cryptographically Secure Signatures
- Can Tolerate Many More Faults

## However

- Questionable Cryptographic Assumptions (Key Distribution, Complexity Questions)
- Traitor May Simply GUESS Generals Key Possible, Though Very Unlikely

## Agreement Using Signed Messages (SM) (Lamport, Shostak, & Pease 1982)

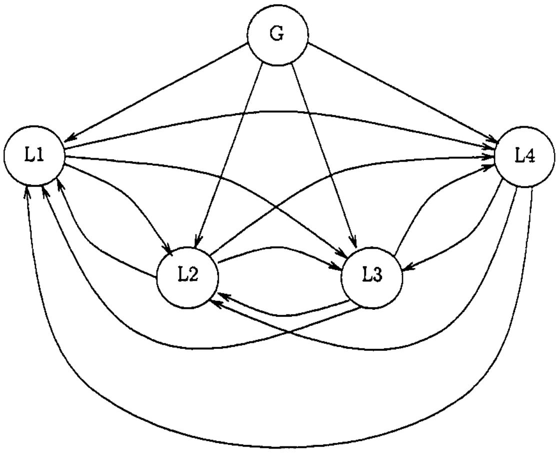
- $r+1$ Rounds Of Message Exchange
- If 1 Properly Signed Message At End, Adopt It Else General Is Faulty, Adopt Default
- Requires Only $n \geq r$ For Agreement and Validity (Best Possible Bound)
- If Signatures Broken, Protocol Fails Horribly
  One Traitor with Key breaks Agreement and Validity for Any Number of Rounds, Any Number of Processors

## Use Signatures In OMH (OMHA) (Gong, Lincoln, Rushby 1995)

- Same as OMH, But at Every Step Check Signatures
- Symmetric, Arbitrary Lieutenants ⇒ Manifest
- Except: Bad Lieutenants Could Send $R(E)$
- Worst Case With Secure Signatures Same As OMH (Worse Than SM)
- If Signatures Broken, Same Tolerance as OMH (Much Better Than SM)

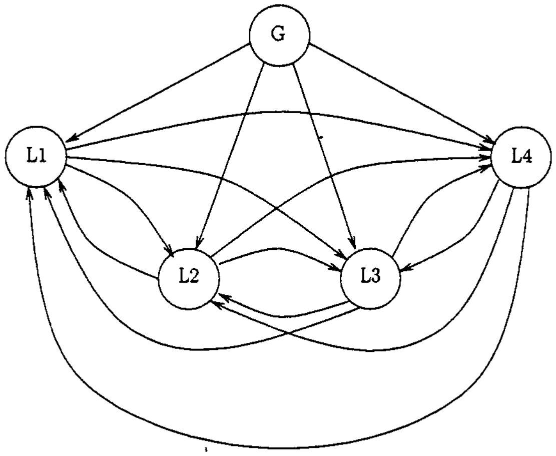
It Would Be Nice to Disallow $R(E)$

## Algorithm OMHA: Signatures Secure



## Algorithm OMHA: Broken Signatures

- $n = 5, r = 1$, the Commander has a Crash Fault, one Lieutenant is a Traitor with Crypto Key
- Good Lieutenants Not Fooled: Protocol Succeeds

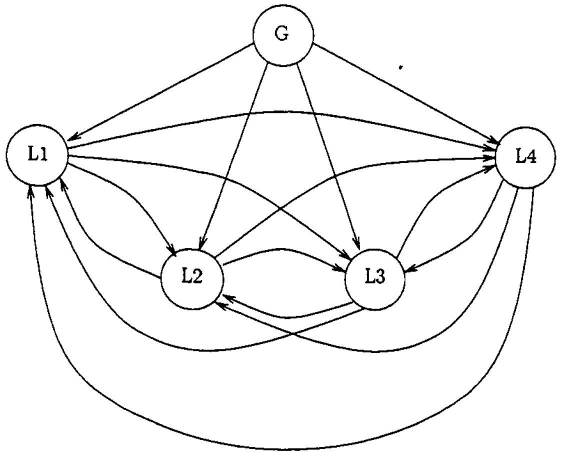


## Use Signatures In Z (ZA) (Gong, Lincoln, Rushby 1995)

- Same as Z, But at Every Step Check Signatures
- Symmetric, Arbitrary Lieutenants ⇒ Manifest
- Bad Lieutenants Cannot Send R(E)
- With Secure Signatures, ZA is Optimal (Same as SM, Better than OMH)
- If Signatures Broken, Same Tolerance as Z (Better Than SM, Worse Than OMH)

## Algorithm ZA: Broken Signatures

- $n = 5$, $r = 1$, the Commander has a Crash Fault, one Lieutenant is a Traitor with Crypto Key
- Good Lieutenants will believe whatever they receive from the Traitor: Protocol Fails



## Symbolic Fault Injection (Gong, Lincoln, Rushby 1995)

- Using Dill's Mur$\phi$ Tool (Stanford)
- Specified OM, OMH, OMHA, SM, Z, ZA
- Complete, Exhaustive State Exploration 5 Processors, 4 Values
- All Above Bounds Correct Rediscovered Flaw in Z

## Symbolic Fault Injection: Beyond The Worst Case Bound

$> 20,000$ Fault Configurations
(Most Beyond Above Bounds)

45

## Clock Synchronization

- All Above Protocols Require Synchronization
- Byzantine Processors May Lie About Clock Value
- Assume Bounded Clock Drift For Good Clocks
- Goal: Maintain Bounded Clock Skew Within $\delta$

46

## Interactive Convergence Algorithm: ICA (Lamport & Melliar-Smith '85)

- Broadcast Current Clock Value
- Compute Egocentric Mean
  - Ignore Values Too Far Off

    n > 3a
- Algorithm Correct
- All But One Lemma Incorrect
- Proof of Main Theorem Incorrect
- Corrected Versions Formally Verified in EHDM
  (Rushby & VonHenke 93) Couple of Months Effort
- Young Formally Verified in NQTHM
  Noted Perfect Clocks Excluded By Rushby
  $<$ should be $\leq$ in Axiom

47

## Other Clock Synchronization Algorithms

- Fred Schneider's General Treatment
- Natarajan Shankar Formally Specified and Verified in EHDM
  Corrected Several Small Flaws
- Verified That ICA Satisfies Corrected Properties
- Paul Miner Found Better, Weaker Assumption Set
  Example Lundelius-Lynch Algorithm

48

## Hybrid Interactive Convergence: ICAH (Rushby 94)

- Ignore E Values In Egocentric Mean

$$n > 3a + 2s + c$$

- Extend Hybrid Fault Model to Link Faults:
  - Model Links Separate From Processors
  - Links Only Introduce Manifestly Bad Values
  - Two Links Between Each Pair
    (In, Out)
  - Asymmetric: Good or Manifest Value

$$n > 3a + 2s + c + l$$

- Formally Specified and Verified in EHDM
  (Rushby 94)
  Couple of Days Effort

## Diagnosis

- Key Part of FDIR:
  Detection, Identification, and Reconfiguration

- Off-Line: After/Between Missions
  - Requires Record Keeping/Testing Harnesses
  - Few Additional Mechanisms In-Flight
  - May Miss Intermittent Faults
  - Use Test Injection Sequences
- On-Line: During Mission
  - Requires Additional Safety-Critical Mechanisms
  - Consumes Mission Resources
  - Survives More Faults During Mission

## Benefits of On-Line Diagnosis

OMH Masks Crashes Already: Why Diagnose?

- Diagnose ALL Symmetric Faults
  Greater Reliability

- Diagnose SOME Byzantine Faults
  Greater Reliability

- Manifest Faults May Become Worse

- Stop Progression:
  Manifest ⇒ Symmetric ⇒ Byzantine

- Restart Crashed Processor

## On-Line Diagnosis

- Easy Without Byzantine Faults

- All Faults Are Symmetric
  Private Accusations Correct

- Impossible To Perfectly Diagnose Byzantine Faults

## Desirable Properties

Correctness (Fairness, Soundness):
Every Processor Diagnosed Faulty IS Faulty

Completeness:
Every Faulty Processor is Diagnosed Faulty

53

## On-Line Byzantine Diagnosis

- PMC, Comparison Testing Inappropriate
- Shin & Ramanathan '87 (Requires Authentication)
- Ramarao & Adams '88 (Optimal, NP-hard)
  Extreme Cost
- Walter '90, Walter, Suri & Hugue '94: Spectrum of Algorithms
  PP, PLP, DD, HD

54

## Algorithm PP: Walter '90

- Assume Symmetric Faults
- Correctness and Completeness
- Formally Specified and Verified in PVS
  (Lincoln 95)
  Couple Weeks Effort

55

## Algorithm PP

- Monitor All Incoming Messages
- For Incorrect Messages, Record Accusation
- At End of Period, Broadcast Accusations
- If $\geq \lceil N/2 \rceil$ Accusations, Declare Faulty
  - Accuse All Non-Accusing Processors
- If $< \lceil N/2 \rceil$ Accusations, Declare NonFaulty
  - Accuse All Accusing Processors
- Iterate

56

## Algorithm PLP: Walter '90

- PP with Link Faults
- Correctness and Partial Completeness
- Formally Specified and Verified in PVS (Lincoln 95)
- Heavy Reuse of Lemmas/Proofs from PP
  Couple Days Effort

## Algorithm PLP

- Like PP
- Except, two kinds of Error:
  Symmetric/Asymmetric
- Based on Kind of Erroneous Message
- Asymmetric Errors Assumed to Be Manifest as Asymmetric-Looking Values

## Algorithm DD: Walter, Suri, & Hugue '94

- True Byzantine Fault Diagnosis
  - Local Detection
  - Byzantine Agreement on Detection: OMH
  - Combine into Diagnosis
- Correctness and Partial Completeness
- Formally Specified and Verified in PVS (Lincoln 95)
- Heavy Reuse of Lemmas/Proofs
  from OMH & PLP
  Couple Hours Effort

## Algorithm DD

- Monitor All Incoming Messages
- For Incorrect Messages, Record Accusation
- Use OMH to Agree on Accusations
- If $\geq \lceil N/2 \rceil$ Accusations, Declare Faulty
- If $< \lceil N/2 \rceil$ Accusations, Declare NonFaulty
- Iterate

## Algorithm HD: Walter, Suri, & Hugue '90

- True Byzantine Fault Diagnosis
- DD with Penalties, Accumulation, Decay
- Correctness and Partial Completeness
- Formally Specified and Verified in PVS (Special Case) (Lincoln 95)
- Heavy Reuse of Lemmas/Proofs From DD Couple Hours Effort

## Algorithm HD

- Like DD
- Except, Different Penalty Weights for Different Errors
- Decay of Accusations
- Cumulative Penalty $> K$, Declare Faulty

## Benefits of Formal Verification

- Identify Exact Assumptions
- Explore Alternatives
- Extremely High Assurance
  - Diagnosis Is Safety-Critical
- Documentation

## Important Properties of User

- Mathematical Sophistication
- Application-Area Knowledge
- Experience

## Important Properties of Proof Checker

- Cite Previous Lemmas and Theorems
- Library Mechanism
- Reuse Parts of Proofs
- Fast Cycle Time (Human-Computer-Human)
- Understandable Output in the Loop
- Heavy Controllable Automation
  (Rewriting, Decision Procedures)
- Exploration Tools
  (Mur$\phi$)

## Bottom Line

Exploration of Algorithms, Assumptions

- Assurance: Formal Verification
- Little Hardware: FTP
- Cheap Reliability: Hybrid Algorithms
  - Byzantine Case Ensures Coverage
  - Simple Cases Handled Cheaper
  - More Reliability With Same Hardware
  - Widely Applicable Paradigm
    Clocks, Agreement, Diagnosis

"The virtue of a logical proof is not that it compels belief but that it suggests doubts" (Lakatos)

## Summary

- Good Hybrid Fault-Tolerant Algorithms
- Informal Proofs are Unreliable
- Real Theorem Proving Still Expensive
- Interesting Examples Now Feasible
- Human-Computer Interaction A Key Element
- Proof Reuse Lowers Cost of Formal Verification

C-3.

# MODEL CHECKING

*David L. Dill*

*Stanford University*

Formal methods have not had the kind of impact we might have hoped. I suggest that the reason is economic: the cost/benefit ratio is unacceptable in many cases, and unproven in most. Hence, research should examine the question of reducing costs, in the form of labor and time.

Automatic formal verification methods for finite-state systems, also known as model-checking, successfully reduce labor costs since they are mostly automatic. Model checkers explicitly or implicitly enumerate the reachable state space of a system, whose behavior is described implicitly, perhaps by a program or a collection of finite automata. Simple properties, such as mutual exclusion or absence of deadlock, can be checked by inspecting individual states. More complex properties, such as lack of starvation, require search for cycles in the state graph with particular properties.

Specifications to be checked may consist of built-in properties, such as deadlock or "unspecified receptions" of messages, another program or implicit description, to be compared with a simulation, bisimulation, or language inclusion relation, or an assertion in one of several temporal logics.

Finite-state verification tools are beginning to have a significant impact in commercial designs. There are many success stories of verification tools finding bugs in protocols or hardware controllers. In some cases, these tools have been incorporated into design methodology.

Research in finite-state verification has been advancing rapidly, and is showing no signs of slowing down. Recent results include probabilistic algorithms for verification, exploitation of symmetry and independent events, and the use symbolic representations for Boolean functions and systems of linear inequalities.

One of the most exciting areas for further research is the combination of model-checking with theorem-proving methods. I will briefly describe some initial forays into this area.

# Model Checking

David L. Dill
Computer Systems Laboratory
Stanford University

Why isn't formal verification used routinely?

- Engineers don't know enough math/logic
- People aren't properly trained
- Bad notation and user interfaces
- Inertia – design methodologies are not easily changed
- Managers don't require it
- Blind stupidity

## The main reason

Economics!

- In most cases, cost vs. benefit is unfavorable, or unproven.
- "Conventional" methods are extremely labor-intensive.
- Furthermore, the labor must be highly skilled.
- In many cases, use of formal methods would *delay* completion of the design.

Design time is often crucial for safety- and life-critical applications.

Most designs are "time-to-market" critical (e.g. cost of delay for an microprocessor release $10 million/week).

## Model checking (finite-state methods)

Most NASA work focusses on very general theorem-proving methods.

Alternative: Use less general, but more automatic, methods.

In the finite-state domain, most verification problems become decidable.

Verification can be done automatically by implicit or explicit state enumeration.

## Model-checking — applications

Finite-state methods have been applied successfully in several domains:

- Protocols (communication, network, cache coherence)
- Hardware state machine comparison
- Hardware controllers
- Asynchronous circuits

There are many success stories about using finite-state verification tools.

In some cases, they have been incorporated into commercial product design methodology.

## Tradeoffs

Advantages of model-checking over theorem-proving:

- Highly automated
- Excellent at diagnosing design errors

Disadvantages of finite-state methods:

- Computational complexity
- Abrupt failure when problem grows
- Limited expressiveness

In reality, model-checking and theorem-proving are complementary techniques.

## Explicit state enumeration

Basic idea: Generate reachable states "on-the-fly," searching for "bad" states.

"bad states" could be *deadlock*, or could violate a *per-state property* (e.g. mutual-exclusion).

## Explicit state search

Behavioral description of the system must give *start states*, and a *next-state generator*, which maps a state to a set of next states.

Generators can be derived from almost any reasonable operational description.

Basic search procedure maintains

- a queue of states to be searched
- a table of states already searched

## Search algorithm

```
While queue is not empty, remove a state $s$ from it.
    Let $Q$ be the set of next states of $s$.
        For each $s' \in Q$
            If $s'$ is not in the state table
                If it is bad, report error and halt
                else enter $s'$ in table and insert in the queue.
```

It is important to stop when error occurs (to avoid generating all states).

When an error occurs, a "counterexample" (path from start state to bad state) can be printed.

Search can be in any order, although breadth-first gives the shortest counterexamples.

## Specifications

- Fixed properties (e.g. deadlock, "unspecified reception")
- Comparison with another description (simulation, bisimulation, language inclusion)
- Temporal logic model checking (especially CTL)

## State explosion problem

Of course, a small behavioral description may give rise a large state space.

This is known as the "state explosion problem." It is the central problem in finite-state verification.

Most problems are at least PSPACE-complete, so a general worst-case solution is probably not available.

There are many methods that have been used to extend the practical boundaries of this method.

## Simple methods

- Omit irrelevant implementation details
- Reduce size of system ("down scaling")
- Focus on finding bugs, not proving correctness
- Use modular/compositional verification

## Advanced methods

A number of sophisticated techniques for coping with the state explosion have emerged in the last few years.

*Hash compaction* — Verify probabilistically by storing a certificate for each state (down to 1 bit), to minimize table size.

*Partial-order methods* — Exploit independent events to reduce number of interleavings that must be modelled.

*Symmetry reduction* — Avoid redundancy from symmetry in the system.

*Symbolic state exploration* — Represent state space symbolically (using Boolean functions, linear inequalities).

## BDD-based verification

BDD-based verification has been so successful that it deserves special attention.

A BDD is a data structure for representing Boolean functions.

Every state is represented as a bit-vector

State is in set iff Boolean function is true.

Breadth-first search can be done using BDD operations.

Sometimes, astronomically large state spaces can be checked with BDDs (Clarke: $10^{1300}$ states).

Sometimes, BDD-based verifiers are worse than explicit methods.

## Verification Systems

Verification systems based on state enumeration have been around at least since 1981.

Some existing systems: SPIN (Holzmann et al. AT&T), COSPAN (Kurshan et al., AT&T), Mur$\phi$ (Dill, et al. Stanford), SMV (Clarke and McMillan, Carnegie-Mellon U.)

There are many success stories of using these systems to verify parts of protocols and hardware designs.

All of these systems are currently in use in industry.

## Future directions

There are many exciting research topics in this area:

- New ideas for avoiding the state explosion
- Alternative symbolic representations
- Verification of real-time and hybrid systems.
- Methods supporting abstraction and refinement
- Extension to infinite-state systems.

## Model checking + theorem proving

One of the most exciting research areas is finding ways to combine finite-state and theorem-proving methods.

A simple idea is to use model-checking first to debug the problem and verify for small systems, then use theorem-proving for the general case.

Example: We described and verified a *distributed list protocol* (central part of a multiprocessor cache consistency algorithm) in Mur$\phi$ for 3 processes.

This helped us to debug the protocol *and verification conditions.*

Automatically translated Mur$\phi$ program to PVS, and verified for $n$ process in PVS.

## Model checking + theorem proving

In some cases, tools can be complementary.

Example: We described the Sparc International's multiprocessor memory consistency models in Mur$\phi$.

Mur$\phi$ model can be used to verify synchronization routines.

PVS used to verify consistency with a logical specification of the same memory model.

## Integration

More ambitiously, finite-state methods can be incorporated into a theorem-proving framework.

Practical use of model-checkers already requires reasoning outside of the finite-state domain.

This reasoning can be a source of errors and omissions, and should be formalized.

Example: A $\mu$-calculus model checker has been embedded in PVS.

When a user generates a lemma in $\mu$-calculus, the model-checker can be called to check it automatically, much more efficiently than general decision procedures of PVS.

There are many opportunities for developing new reduction strategies.

## Conclusions

- Economics are crucial.
- Use of verification for finding bugs may be more important than correctness proofs.
- Formal verification based on finite-state techniques is already successful, and is improving rapidly.
- Combining model-checking and theorem-proving may lead to necessary breakthroughs in verification productivity.

# Session 10: Research Issues (3)

## Ricky W. Butler, Chair

- **The DDD Scheme Machine**, by *Steve Johnson*, Indiana University
- **Formal Development of a Clock Synchronization Circuit**, by *Paul Miner*, NASA Langley Research Center

# The Scheme Machine:
# A Case Study in Progress in Design Derivation at System Levels

*Steven D. Johnson*

*Indiana University*

The Scheme Machine is one of several design projects of the Digital Design Derivation group at Indiana University. It differs from the other projects in its focus on issues of system design and its connection to surrounding research in programming language semantics, compiler construction, and programming methodology underway at Indiana and elsewhere. The genesis of the project dates to the early 1980s, when digital design derivation research branched from the surrounding research effort in programming languages. Both branches have continued to develop in parallel, with this particular project serving as a bridge. However, by 1990 there remained little real interaction between the branches and recently we have undertaken to reintegrate them.

On the software side, researchers have refined a mathematically rigorous (but not mechanized) treatment starting with the fully abstract semantic definition of Scheme and resulting in an efficient implementation consisting of a compiler and virtual machine model, the latter typically realized with a general purpose microprocessor. The derivation includes a number of sophisticated factorizations and representations and is also deep example of the underlying engineering methodology.

The hardware research has created a mechanized algebra supporting the tedious and massive transformations often seen at lower levels of design. This work has progressed to the point that large scale devices, such as processors, can be derived from first-order finite state machine specifications. This is roughly where the language oriented research stops; thus, together, the two efforts establish a thread from the highest levels of abstract specification to detailed digital implementation.

The Scheme Machine project challenges hardware derivation research in several ways, although the individual components of the system are of a similar scale to those we have worked with before. The machine has a custom dual-ported memory to support garbage collection. It consists of four tightly coupled processes---processor, collector, allocator, memory---with a very non-trivial synchronization relationship. Finally, there are deep issues of representation for the run-time objects of a symbolic processing language.

The research centers on verification through integrated formal reasoning systems, but is also involved with modeling and prototyping environments. Since the derivation algebra is based on an executable modeling language, there is opportunity to incorporate design animation in the design process. We are looking for ways to move smoothly and incrementally from executable specifications into hardware realization. For example, we can run the garbage collector specification, a Scheme program, directly against the physical memory prototype, and similarly, the instruction processor model against the heap implementation.

# The Scheme Machine:

**A Case Study in Progress**
**In Design Derivation at System Levels**

Steven D. Johnson
DDD Project, Hardware Methods Laboratory
Computer Science Department
Indiana University

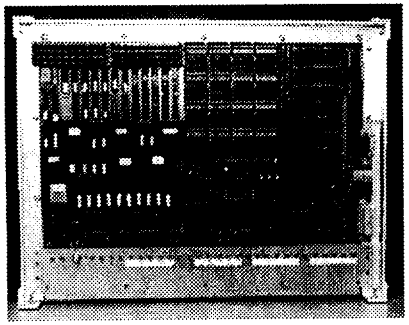
sjohnson@cs.indiana.edu
http://www.cs.indiana.edu/hmg/hmg.html

Thanks: NSF/MIP92 08745, NASA/NGT-50861

## Outline

- The Scheme Machine Prototype
- Context, Motivation, and Goals of the Research
- Components of the Scheme Machine
- Issues in Verification
- Incremental Construction of Hardware
- Summary and Status

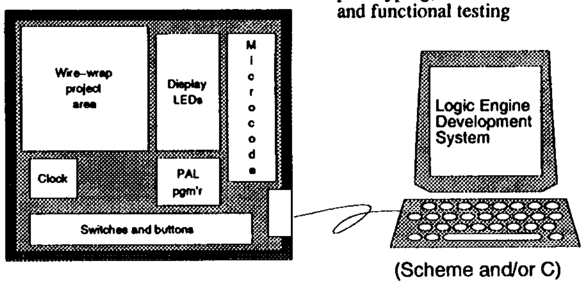
## The Scheme Machine Prototype



Emulator in Actel FPGAs, PLDs and standard DRAM simms.

## The Logic Engine

An environment for sytem prototyping, VLSI emulation, and functional testing

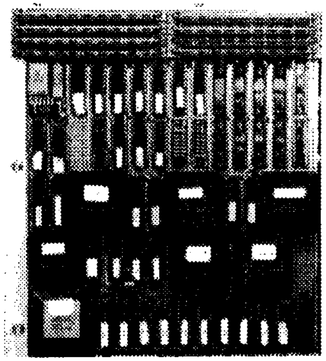


**Scheme Machine**

Memory:
2 × 4 × 1, 000, 000 bytes

Design Size:
20,000 gates (?)

Speed:
2 MHz+ ( or 1.25 × DRAM)



## Context of the Research

1982 M. Wand, *Semantics-directed machine architecture* (POPL 92)

1983 S.D. Johnson, *Synthesis of Digital Designs from Recursion Equations*, Ch. 5 (PhD dissertation)

1984 W. Clinger, *The Scheme 311 Compiler* (L&FP 84)

1987 S.D. Johnson, et. al. *A Tactical Framework for Digital Design* (WG10.2 FM/VLSI).

1988 R. Wehrmeister, *Derivation of an SECD Machine* (IU/CS TR #290)

1990 D. Boyer and K. Rath derive boolean system for a Scheme machine.

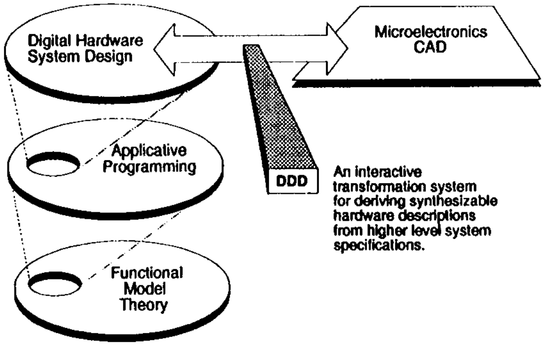
1991 IEEE Std 1178-1990 *IEEE Standard for the Scheme Programming Language.*

1992 M. Wand, D. Friedman, C. Haynes, *Essentials of Programming Languages*, Ch. 12.

1993 B. Burger, *The Scheme Machine* (IU/CS TR #413).

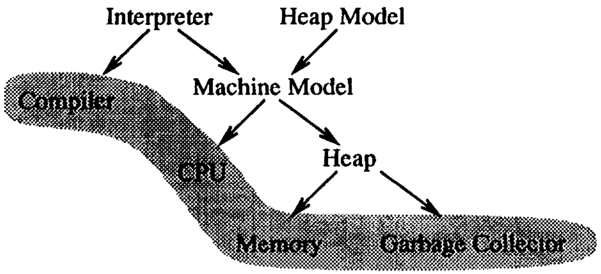
1995 M. Wand, J. Guttman, J. Ramsdell, et.al. *VLISP: A Verified Implementation of Scheme* (*J. Lisp and Symbolic Computation*).

## Digital Design Derivation Project

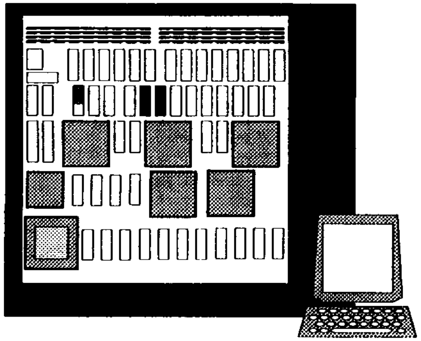


An interactive transformation system for deriving synthesizable hardware descriptions from higher level system specifications.

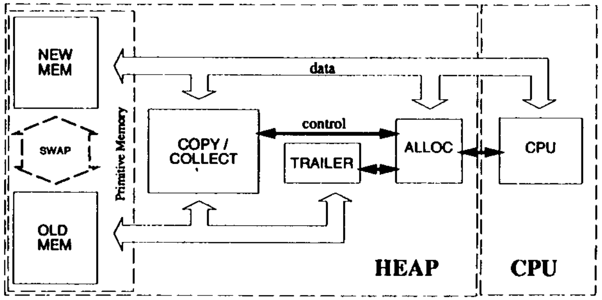
## The Scheme Machine Project



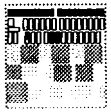
Scheme Machine Prototyping Environment



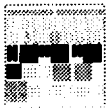
Scheme Machine Architecture



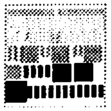
## Components of the Scheme Machine



*Memory*
o Dual-ported semispaces, multiplexed bus
o typical case: 2 opns. per 1.25 cycles
o refresh inhibits clock
o manual design, unverified (one bug so far)

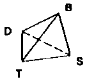


*Heap Interface*
o collector, allocator, invalidator
o derived from XFSM.
o synchronization with CPU not verified
o algorithm not verified (VLISP?)



*Processor*
o derived from X-FSM only.
o SPEC close to VLISP machine?

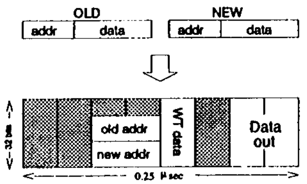
## Issues in Verification



- Higher level behavior specification
- Memory model
- Process synchronization
- Data abstraction hierarchy
- Multiple views

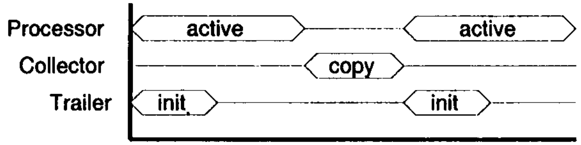
## Memory Model

Memory Interface



```
rd(N, a)     || wt(O, b,c)
wt(N, a, b) || rd(O, c)
rd(N, a)     || mark-next(O)
wt(N, a, b) || mark-next(O)
```

## Process Synchronization



- In Scheme Machine, heap integrity is guaranteed by hardware, with overhead absorbed (to be measured) by the parallel sweep.
- In VLISP, heap integrity is guaranteed by the compiler, but is interruption considered?

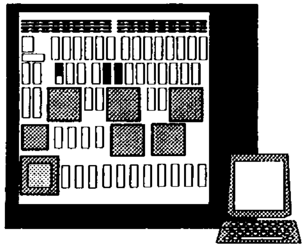
## Data Abstraction

- semantic model: $\rho$: *Var* → *Val*
- compiler factorization: $\rho_c$: *Var* → *StaticOffset*
  $\rho_r$: *StaticOffset* → *Store* → *Val*
- Operational model: Rep[$\rho_r$] = *List(Vector)*
- Implementation model: *List*: *HeapReference*
  *Vector*: (*HeapReference*, *Offset*)
- Representation: *HeapReference*: (*Tag*, *Address*)
  *Offset*: *Address*
  *Tag*: {list, int, . . .}
  *Address*: $\mathcal{Z}_{2^{24}}$
- HW Realization: *Address*, *Offset*: *BitVector*(24)
  *Tag*: *BitVector*(8)

## Multiple Views

- The CPU sees the heap as an assorted collection of cells, including numbers, list cells of various flavors, vectors, strings, *etc.*
- The allocator and collector see the heap as an array of words subject to certain invariants.
- The invalidator sees the heap as a sequence of bits.

## Incremental Construction of Hardware



## Incremental Construction of Hardware

- The software specification of the processor runs directly against both the software specification of the heap and (with functional test points exposed) the hardware prototype.
- The software specification of the processor, allocator, and collector, run directly against both the modeling environment's heap image and the hardware realization.
- We are seeking to develop an environment where modeling, simulation, emulation, and realization are closely integrated with verification.
- We are seeking to integrate multiple reasoning tools to apply to the verification problem.

## Summary and Status

- Design derivation extends and adapts programming methodology to hardware targets
- Language implementation, Scheme in this case, is a standard example,
  - exposes hard (higher order) modeling issues
  - a form of closure/completeness (Scheme Machine derivation & emulation could be run on the Scheme Machine).
- the Scheme Machine system specification exposes new verification problems in process coordination, algorithmic correctness, ....
- the design environment explores the gap between modeling and implementation.
- components of the Scheme Machine, or their variations are viable products.

# Formal Development of a Clock Synchronization Circuit

Paul S. Miner

This talk presents the latest stage in a formal development of a fault-tolerant clock synchronization circuit. The development spans from a high level specification of the required properties to a circuit realizing the core function of the system.

An abstract description of an algorithm has been verified to satisfy the high-level properties using the mechanical verification system EHDM [2]. This abstract description is recast as a behavioral specification input to the Digital Design Derivation system (DDD) developed at Indiana University [1]. DDD provides a formal design algebra for developing correct digital hardware. Using DDD as the principle design environment, a core circuit implementing the clock synchronization algorithm was developed [3]. The design process consisted of standard DDD transformations augmented with an ad hoc refinement justified using the Prototype Verification System (PVS) from SRI International [4].

Subsequent to the above development, Wilfredo Torres-Pomales discovered an area-efficient realization of the same function [5]. Establishing correctness of this optimization requires reasoning in arithmetic, so a general verification is outside the domain of both DDD transformations and model-checking techniques.

DDD represents digital hardware by systems of mutually recursive stream equations. A collection of PVS theories was developed to aid in reasoning about DDD-style streams. These theories include a combinator for defining streams that satisfy stream equations, and a means for proving stream equivalence by exhibiting a stream bisimulation.

DDD was used to isolate the sub-system involved in Torres-Pomales' optimization. The equivalence between the original design and the optimized verified was verified in PVS by exhibiting a suitable bisimulation. The verification depended upon type constraints on the input streams and made extensive use of the PVS type system. The dependent types in PVS provided a useful mechanism for defining an appropriate bisimulation.

## References

[1] Bhaskar Bose. DDD - A Transformation System for Digital Design Derivation. Technical Report 331, Computer Science Dept. Indiana University, May 1991.

[2] Paul S. Miner. Verification of fault-tolerant clock synchronization systems. Technical Paper 3349, NASA, Langley Research Center, Hampton, VA, November 1993.

[3] Paul S. Miner, Shyamsundar Pullela, and Steven D. Johnson. Interaction of formal design systems in the development of a fault-tolerant clock synchronization circuit. In *Proceedings 13th Symposium on Reliable Distributed Systems*, pages 128–137, Dana Point, CA, October 1994.

[4] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[5] Wilfredo Torres-Pomales. An optimized implementation of a fault-tolerant clock synchronization circuit. Technical Memorandum 109176, NASA, Langley Research Center, Hampton, VA, February 1995.

# Formal Development of a Fault-Tolerant Clock Synchronization Circuit

Paul S. Miner
May 12, 1995

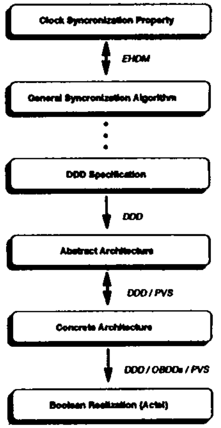
## Outline

- Summary of Prior work
- Description of Torres-Pomales' Optimization
- Verification of Optimization
  - Definition of Streams in PVS
  - Proof by Co-Induction

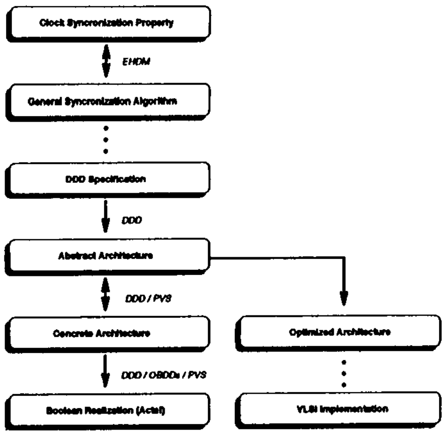
## Prior Verification

- Developed verified design of clock synchronization circuit using a combination of formal techniques.
  - Mechanized Proof System (EHDM, PVS)
  - Digital Design Derivation
  - OBDD-based tautology checking

## Design Hierarchy—Old


Clock Syncronization Property
EHDM
General Syncronization Algorithm
DDD Specification
DDD
Abstract Architecture
DDD / PVS
Concrete Architecture
DDD / OBDDs / PVS
Boolean Realization (Actel)

## Design Hierarchy—New



Clock Syncronization Property

EHDM

General Syncronization Algorithm

DDD Specification

DDD

Abstract Architecture

DDD / PVS

Concrete Architecture

DDD / OBDDs / PVS

Boolean Realization (Actel)

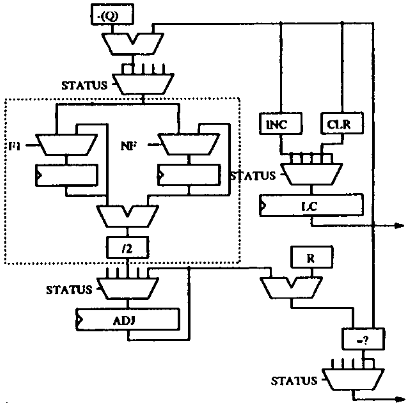
Optimized Architecture

VLSI Implementation

## Informal Description of Algorithm

- Welch & Lynch Algorithm
- System of $N$ clocks designed to tolerate $F$ arbitrary faults
- Completely connected network
- Each Clock periodically
  - Gathers estimates of readings of all other clocks in the system
  - Discards the $F$ largest and $F$ smallest readings
  - Sets self to mid-point of the range of the remaining readings

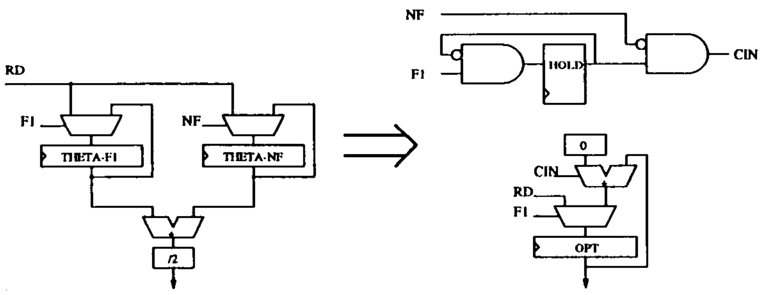
## Intermediate Stage of Previous Derivation



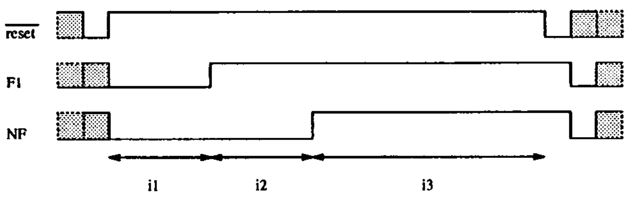
## Intermediate Stage

- Circuit implements core function of algorithm
  - Network interconnect in different partition of design
- Independent of number of clocks in the system
- This stage was reached via a combination of standard DDD transformations and an *ad hoc* refinement verified using PVS

## Torres-Pomales' Optimization



## Signal Assumptions Justifying Optimization



Signal RD is the output of a counter.

## Verification of Optimized Circuit

- Reasoning about Stream Equations using PVS
  - Definition of Streams in PVS
  - Proof by Co-Induction
- Verification Using PVS Streams Package

## Streams in PVS

DECLARATIONS

```
Stream_adt[alpha: TYPE]: THEORY
  BEGIN

  Stream: TYPE

  a: VAR alpha
  S, X, Y: VAR Stream

  cs?: [Stream -> boolean]

  cs: [alpha, Stream -> Stream]
  hd: [Stream -> alpha]
  tl: [Stream -> Stream]

  nth(S:Stream,n:nat):alpha = hd(iterate(tl,n)(S))
```

## Streams in PVS

AXIOMS

```
  Stream_inclusive: AXIOM cs?(S)

  Stream_cs_eta: AXIOM cs(hd(S), tl(S)) = S

  Stream_hd_cs: AXIOM hd(cs(a, S)) = a

  Stream_tl_cs: AXIOM tl(cs(a, S)) = S

  Stream_eq: AXIOM X = Y <=> FORALL n: nth(X, n) = nth(Y, n)

END Stream_adt
```

## Defining Streams

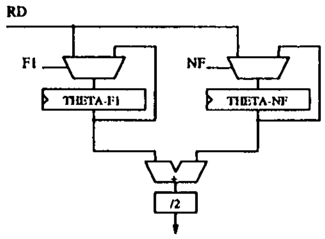
```
Stream_corec[alpha, beta: TYPE]: THEORY
  BEGIN
  IMPORTING Stream_adt[beta]

  f: VAR [alpha -> beta]
  g: VAR [alpha -> alpha]
  a: VAR alpha

  corec(f, g, a): Stream[beta]

  corec_def: AXIOM corec(f, g, a) = cs(f(a), corec(f, g, g(a)))

  [...]

  END Stream_corec
```

## Proof by Co-Induction

```
Stream_coinduct[alpha: TYPE]: THEORY
  BEGIN

  IMPORTING Stream_adt

  X, Y: VAR Stream[alpha]

  R: VAR PRED[[Stream[alpha], Stream[alpha]]]

  Bisimulation: TYPE =
    {R | FORALL X, Y: R(X, Y) => hd(X) = hd(Y) & R(tl(X), tl(Y))}

  co_induct: THEOREM (EXISTS (R: Bisimulation): R(X, Y)) => X = Y

  END Stream_coinduct
```

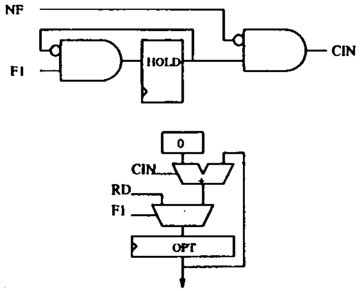
## Stream Equations for Original Sub-Circuit

$$\begin{aligned}
\texttt{THETA-F1} &= \texttt{cs}(i,\texttt{MUX(F1, RD, THETA-F1)}) \\
\texttt{THETA-NF} &= \texttt{cs}(i,\texttt{MUX(NF, RD, THETA-NF)}) \\
\texttt{CFN} &= \left\lfloor \frac{\texttt{THETA-F1} + \texttt{THETA-NF}}{2} \right\rfloor
\end{aligned}$$



## Stream Equations for Optimized Sub-Circuit

```
HOLD = cs(false, F1 & ¬HOLD)
 CIN = HOLD & ¬NF
 OPT = cs(i,MUX(F1, RD, INC(OPT,CIN)))
```

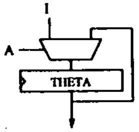


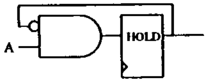
## PVS Definitions for Circuit Verification

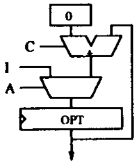
```
A,B,C,R: VAR Stream[bool]
a,b,c,r: VAR bool
I,J,K: VAR Stream[int]
i,j,k: VAR int

THETA(A,I,i): Stream[int]    %defined using corec

CFN(A,B,I,i,j): Stream[int]
        = DIV2(THETA(A,I,i) + THETA(B,I,j))

HOLD(A,a): Stream[bool]       %defined using corec

CIN(A,B): Stream[bool] = A AND NOT B

OPT(A,C,I,i): Stream[int]    %defined using corec
```

## Recursive Stream Definitions

```
THETA(A,I,i) = cs(i,MUX(A,I,THETA(A,I,i)))
  HOLD(A,a) = cs(a, A & ¬HOLD(A,a))
OPT(A,C,I,i) = cs(i, MUX(A,I,INC(OPT(A,C,I,i),C) ))
```







## Type Declarations for Assumptions on Input Signals

```
S(R): TYPE =
  {A |Invariant(IF R
                  THEN NOT tl(A)
                  ELSE A ⇒ tl(A)
                  ENDIF)}

C(R): TYPE =
  {I| Invariant(NOT R ⇒ EQ(tl(I),INC(I)))}
```

## Correctness Theorem

```
Optimize_correct: THEOREM

  ∀ R,(RD : C(R)),(F1 : S(R)| ¬hd(F1)),
     (NF : S(R)|Invariant(NF ⇒ F1)),(i : int):

   CFN(F1,NF,RD,i,i) = OPT(F1,CIN(HOLD(F1,false),NF),RD,i)
```

## Proof of Optimize_correct by co-induction

Define Bisimulation $B$ as:

```
{(X,Y)|
   ∃ R,(RD : C(R)),(F1 : S(R)),(NF : {A : S(R)|A ⇒ F1}),(i : int)
      (j : int|hd(F1) ∧ ¬(hd(NF)) ⇒  hd(RD) = j + 1),
        (b : bool|hd(F1) ∧ ¬(hd(NF)) ⇒  b = odd?(i + j)) :
   X = CFN(F1,NF,RD,i,j) &
   Y = OPT(F1,CIN(HOLD(F1,b),NF),RD,⌊(i + j)/2⌋)}
```

## Proof—$B$ is a Bisimulation

**Heads:** For any $(X,Y) \in B$, hd($X$) = hd($Y$) = $\lfloor (i+j)/2 \rfloor$.

**Tails:** For any $(X,Y) \in B$, show (tl($X$),tl($Y$))$\in B$.

```
tl(CFN(F1,NF,RD,i,j))
      = CFN(tl(F1),tl(NF),tl(RD),
            IF hd(F1) THEN i ELSE hd(RD) ENDIF,
            IF hd(NF) THEN j ELSE hd(RD) ENDIF)

tl(OPT(F1,CIN(HOLD(F1,b),NF),RD,⌊(i + j)/2⌋)))
    = OPT(tl(F1),
          CIN(HOLD(tl(F1), (hd(F1)∧ ¬ b))),tl(NF)),
          tl(RD),
          IF hd(F1) THEN (⌊(i + j)/2⌋) + [b ∧ ¬hd(NF)]
          ELSE hd(RD)
          ENDIF)
```

## Concluding Remarks

- Proof by co-induction effective technique for verifiying circuit refinements.
  - Possible to exploit circuit context to complete proof
- Developed general Stream library for PVS 2
- Torres-Pomales' optimization verified in PVS using proof by co-induction
- PVS dependent type mechanism useful
- Design implemented in VLSI (hand layout)

Omit to
END

# Appendix A:

## List of Attendees

# ATTENDEE LIST

## Third NASA Langley Formal Methods Workshop
May 10-12, 1995

Abernethy, Ken
Furman University
Dept. of Computer Science
Greenville, SC 29613
Email: aberneth@furman.edu
Phone: 803-294-3219
Fax: 803-294-2058

Aboulhamid, El Mostapha
Universite de Montreal, Dept. IRO
C.P. 6128, Suc. Centre Ville
Montreal, QC H3C3J7
Email: aboulham@iro.umontreal.ca
Phone: 514-343-6822
Fax: 514-343-5834

Abraham, Jacob A.
University of Texas at Austin
Comp. Eng. Research Ctr.
ENS 424, MC C8800
Austin, TX 78712
Email: jaa@cerc.utexas.edu
Phone: 512-471-8983
Fax: 512-471-8967

Andrianos, Nikos P.
Union Switch and Signal, Inc.
1000 Technology Drive
Pittsburgh, PA 15219-3120
Email: npa@switch.com
Phone: 412-934-2138
Fax: 412-934-2190

Baraona, Phillip
University of Cincinnati
Elect. & Comp. Eng.
Cincinnati, OH 45221-0030
Email: pbaraona@ece.uc.edu
Phone: 513-579-0614
Fax: none given

Bettis, Robert N.
Harmon Electronics, Inc.
PO Box 600
Grain Valley, MO 64029
Email: rbettis@tyrell.net
Phone: 816-650-6171 ext. 616
Fax: 816-650-3383

Bose, Bhaskar
Derivation Systems, Inc.
5963 La Place Ct.
Suite 208
Carlsbad, CA 92008
Email: bose@dvsi.com
Phone: 619-431-1400
Fax: 619-431-1484

Boxer, Alan
Department of Defense, Attn: C75
9800 Savage Rd.
Ft. Meade, MD 20755-6000
Email: boxer@aztech.ba.md.us
Phone: 301-688-4229
Fax: 301-688-2080

Brill, Robert
NRC, T-10-E33
11545 Rockville Pike
Rockville, MD 20852-2738
Email: rwb2@nrc.gov
Phone: 301-415-6760
Fax: 301-415-5160

Brock, Bishop
Computational Logic, Inc.
1717 West 6th St., Suite 290
Austin, TX 78703
Email: brock@cli.com
Phone: 512-322-9951
Fax: 512-322-0656

Butler, Ricky W.
NASA Langley Research Ctr.
Mail Stop 130
Hampton, VA 23681-0001
Email: r.w.butler@larc.nasa.gov
Phone: 804-864-6198
Fax: 804-864-4234

Caldwell, James L.
NASA Langley Research Center
Mail Stop 130
Hampton, VA 23681-0001
Email: jlc@air16.larc.nasa.gov
Phone: 804-864-6214
Fax: 804-864-4234

Carreno, Victor A.
NASA Langley Research Center
Mail Stop 130
Hampton, VA 23681-0001
Email: vac@air16.larc.nasa.gov
Phone: 804-864-6184
Fax: 804-864-4234

Charlesworth, Art
University of Richmond
Mathematics & Computer Science
Richmond, VA 23173
Email: charlesworth@mathcs.urich.edu
Phone: 804-289-8636
Fax: 804-287-6444

Chin, Shiu-Kai
Rome Laboratory, C3AB
525 Brooks Rd.
Griffiss AFB, NY 13441
Email: chin@cliff.rl.af.mil
Phone: 315-330-3241
Fax: 315-330-2807

Danner, Bonnie P.
TRW SIG, WDC9/7S184
One Federal Systems Park Drive
Fairfax, VA 22033-4416
Email: Bonnie_Danner_at_SETA.@mail.hq.faa.gov
Phone: 202-651-2254
Fax: none given

Delisle, Norman
Raytheon Company, MS 3-2-3925
1001 Boston Post Rd.
Marlborough, MA 01752
Email: normd@sud.ed.ray.com
Phone: 508-490-2771
Fax: 508-490-1103

Di Vito, Ben L.
Vigyan, Inc.; NASA-LaRC
Mail Stop 130
Hampton, VA 23681-0001
Email: b.l.divito@larc.nasa.gov
Phone: 804-864-4883
Fax: 804-864-4234

Dill, David L.
Stanford University, CIS 135
Stanford, CA 94305-4070
Email: dill@cs.stanford.edu
Phone: 415-725-3642
Fax: 415-725-6278

Eckhardt, Dave E.
NASA Langley Research Center
Mail Stop 152D
Hampton, VA 23681-0001
Email: d.e.eckhardt@larc.nasa.gov
Phone: 804-864-1698
Fax: 804-864-8672

El-Agamawi, Mohsen M.
Union Switch and Signal, Inc.
1000 Technology Drive
Pittsburgh, PA 15219-3120
Email: mme@switch.com
Phone: 412-934-2112
Fax: 412-934-2190

Elks, Carl R.
US Army CECOM
Mail Stop 152D
Hampton, VA 23681-0001
Email: cre@air16.larc.nasa.gov
Phone: 804-864-4882
Fax: 804-864-8672

Fisler, Kathi
Indiana University
Dept. of Computer Science
Lindley Hall 215
Bloomington, IN 47405
Email: kfisler@cs.indiana.edu
Phone: 812-855-3652
Fax: 812-855-4829

Fleming, David
West Virginia University
Dept. of Statistics & CS
Morgantown, WV 26505
Email: fleming@cs.wvu.edu
Phone: 304-293-3607
Fax: none given

Fletcher, Sharon K.
Sandia National Laboratories
Org. 9411 (MS-0777)
PO Box 5800
Albuquerque, NM 87185-0777
Email: skfletc@sandia.gov
Phone: 505-844-2251
Fax: 505-844-9641

French, Scott W.
LORAL--Air Traffic Control
9231 Corporate Blvd.
Bldg. 861/4C08
Rockville, MD 20850
Email: none given
Phone: 301-640-2685
Fax: none given

Goddard, Peter
Hughes Aircraft Company
PO Box 3310
M/S 618/P311
Fullerton, CA 92634
Email: pgoddard@msmail2.hac.com
Phone: 714-732-7754
Fax: 714-732-2613

Goel, Amrit L.
Syracuse University
5011 Woodside Rd.
Fayetteville, NY 13066
Email: goel@cat.sys.edu
Phone: 315-443-4350
Fax: 315-443-2583

Guaspari, David
Odyssey Research Associates
301 Dates Drive
Ithaca, NY 14850
Email: davidg@oracorp.com
Phone: 607-277-2020
Fax: 607-277-3206

Hayhurst, Kelly J.
NASA Langley Research Center
Mail Stop 159
Hampton, VA 23681-0001
Email: k.j.hayhurst@larc.nasa.gov
Phone: 804-864-6215
Fax: 804-864-9713

Holloway, C. Michael
NASA Langley Research Center
Mail Stop 130
Hampton, VA 23681-0001
Email: c.m.holloway@larc.nasa.gov
Phone: 804-864-1701
Fax: 804-864-4234

Holt, H. Milton
NASA Langley Research Center
Mail Stop 130
Hampton, VA 23681-0001
Email: h.m.holt@larc.nasa.gov
Phone: 804-864-1596
Fax: 804-864-8821

Hoover, Doug
Odyssey Research Associates
301 Dates Drive
Ithaca, NY 14850
Email: hoove@oracorp.com
Phone: 607-277-2020
Fax: 607-277-3206

Huggins, James K.
Univ. of Michigan--EECS Dept.
3114 EECS Bldg.
1301 Beal Ave.
Ann Arbor, MI 48109-2122
Email: huggins@umich.edu
Phone: 313-763-4526
Fax: 313-763-1503

Hugue, Michelle McElvany
Opsimath Research
2521 Kitmore Lane
Bowie, MD 20715-2751
Email: meesh@cs.umd.edu
Phone: 301-262-5140
Fax: 301-262-5988

Hurst, Sandra C.
NASA Langley Research Center
Mail Stop 131
Hampton, VA 23681-0001
Email: s.c.hurst@larc.nasa.gov
Phone: 804-864-6219
Fax: 804-864-7794

Jackson, Paul
Cornell University
Dept. of Computer Science
Upson Hall
Ithaca, NY 14853
Email: jackson@cs.cornell.edu
Phone: 607-255-6046
Fax: 607-255-4428

Jamsek, Damir
Odyssey Research Associates
301 Dates Drive
Ithaca, NY 14850
Email: damir@oracomp.com
Phone: 607-277-2020
Fax: 607-277-3206

Janeri, John
Underwriters Laboratories, Inc.
12 Laboratory Drive
PO Box 13995
Research Triangle Park, NC 27709-3995
Email: janeri@ul-rtp.interpath.net
Phone: 919-549-1902
Fax: 919-549-1452

Johnson, Sally C.
NASA Langley Research Center
Mail Stop 248
Hampton, VA 23681-0001
Email: sally.c.johnson@larc.nasa.gov
Phone: 804-864-6204
Fax: 804-864-3553

Johnson, Steven D.
Indiana University, Computer Science Dept.
Lindley Hall 215
Bloomington, IN 47405
Email: sjohnson@cs.indiana.edu
Phone: 812-855-4510
Fax: 812-855-4829

Kelly, John
NASA JPL
4800 Oak Grove Drive
Pasadena, CA 91109
Email: jkelly@spa1.jpl.nasa.gov
Phone: 818-354-4495
Fax: 818-393-1362

Knight, John C.
University of Virginia
Dept. of Computer Science
Thornton Hall
Charlottesville, VA 22903
Email: knight@virginia.edu
Phone: 804-982-2216
Fax: 804-982-2214

Krishnamurthi, Shriram
Rice University
Dept. of Computer Science
MS 132, 6100 S. Main
Houston, TX 77005-1892
Email: shriram@cs.rice.edu
Phone: 713-523-7964
Fax: 713-285-5930

Kuhn, Rick
Nat'l Inst. of Standards & Technology
Technology Bldg. B266
Gaithersburg, MD 20899
Email: kuhn@nist.gov
Phone: 301-975-4601
Fax: 801-948-7242

Leathrum, James
Old Dominion University
Dept. of Electrical & Comp. Eng.
231 Kaufman-Duckworth Hall
Norfolk, VA 23529
Email: leathrum@ee.odu.edu
Phone: 804-683-3741
Fax: 804-683-3220

Legato, Wilfred
Department of Defense, Attn: C6
9800 Savage Rd.
Ft. Meade, MD 20755-6000
Email: legato@romulus.ncsc.mil
Phone: 301-688-4229
Fax: none given

Lincoln, Patrick
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025
Email: lincoln@csl.sri.com
Phone: 415-859-5454
Fax: 415-859-2844

Madhav, Neel
State University of New York at Binghamton
T20 Watson School
Binghamton, NY 13902
Email: madhav@cs.binghamton.edu
Phone: 607-777-4880
Fax: 607-777-4000

Malekpour, Mahyar
Lockheed/NASA-LaRC
Mail Stop 152D
Hampton, VA 23681-0001
Email: m.r.malekpour@larc.nasa.gov
Phone: 804-864-1513
Fax: 804-864-4234

Miller, Steven P.
Collins Commercial Avionics
Rockwell International
400 Collins Rd. NE
Cedar Rapids, IA 52498
Email: spmiller@pobox.cca.rockwell.com
Phone: 319-395-8008
Fax: 319-395-4068

Miner, Paul S.
NASA Langley Research Center
Mail Stop 130
Hampton, VA 23681-0001
Email: p.s.miner@larc.nasa.gov
Phone: 804-864-6201
Fax: 804-864-4234

Nassif, Michael P.
Rome Laboratory IERDD
525 Brooks Rd.
Griffiss AFB, NY 13441-4505
Email: nassifm@rl.af.mil
Phone: 315-330-3342
Fax: 315-330-2885

Neumann, Peter G.
SRI International
Computer Science Lab; EL-243
Menlo Park, CA 94025-3493
Email: neumann@csl.sri.com
Phone: 415-859-2375
Fax: 415-859-2844

Peckham, Lisa F.
NASA Langley Research Center
Mail Stop 130
Hampton, VA 23681-0001
Email: l.f.peckham@larc.nasa.gov
Phone: 804-864-6220
Fax: 804-864-4234

Penix, John
University of Cincinnati
Electrical & Comp. Eng.
Mail Location #30
Cincinnati, OH 45219
Email: jpenix@ece.uc.edu
Phone: 513-531-8525
Fax: none given

Platek, Richard
Odyssey Research Associates
301 Dates Drive
Ithaca, NY 14850
Email: richard@oracorp.com
Phone: 607-277-2020
Fax: 607-277-3206

Profeta, Joseph A.
Union Switch and Signal, Inc.
1000 Technology Drive
Pittsburgh, PA 15219-3120
Email: jap@switch.com
Phone: 412-934-2186
Fax: 412-934-2190

Quach, Patrick
Lockheed/NASA-LaRC
Mail Stop 130
Hampton, VA 23681-0001
Email: c.c.quach@larc.nasa.gov
Phone: 804-864-7641
Fax: 804-864-4234

Radley, Charles F.
Raytheon Corp.
2001 Aerospace Parkway
Brookpark, OH 44142
Email: rqcfr@lims01.lerc.nasa.gov
Phone: 216-977-1492
Fax: 216-977-1495

Raheja, Dev G.
Technology Management, Inc.
9811 Mallard Drive
Suite 213
Laurel, MD 20708
Email: none given
Phone: 410-792-0710
Fax: 301-953-2213

Reesman, Leslie
Hernandez Eng., Inc; NASA-LaRC
Mail Stop 467
Hampton, VA 23681-0001
Email: none given
Phone: 804-864-9409
Fax: 804-864-8574

Roediger, Paul A.
US Army ARDEC
AMSTA-AR-QAW-P
Picatinny Arsenal, NJ 07806-5000
Email: fuzzy@qa.pica.army.mil
Phone: 201-724-3008
Fax: 201-724-4026

Rushby, John M.
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
Email: rushby@csi.sri.com
Phone: 415-859-5456
Fax: 415-859-2844

Saraceni, Pete
FAA Technical Center
AAR-421 Bldg. 210
Atlantic City Int'l. Airport, NJ 08405
Email: saracenp@admin.tc.faa.gov
Phone: 609-485-5577
Fax: 609-485-4005

Saydjari, Fay F.
Dept. of Defense
309 Eagle Hill Road
Pasadena, MD 21122
Email: ffs@tycho.ncsc.mil
Phone: 410-360-4116
Fax: 301-688-0255

Shankar, Natarajan
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
Email: shankar@csl.sri.com
Phone: 415-859-5272
Fax: 415-859-2844

Sheldon, Rick
Univ. of Texas at Arlington
Computer Sci. & Eng.
916 Yates Ave, Box 19015
Arlington, TX 76019-0015
Email: sheldon@cse.uta.edu
Phone: 817-273-3629
Fax: 817-273-3784

Sherry, Lance
Honeywell--Air Transport Systems
PO Box 21111
Phoenix, AZ 85036
Email: sherry@p15.az75.az.honeywell.com
Phone: 602-436-1274
Fax: 602-436-5151

Shipman, Floyd
NASA Langley Research Center
Mail Stop 130
Hampton, VA 23681-0001
Email: f.s.shipman@larc.nasa.gov
Phone: 804-864-1706
Fax: 804-864-4234

Smith, Kathryn A.
NASA Langley Research Center
Mail Stop 130
Hampton, VA 23681-0001
Email: k.a.smith@larc.nasa.gov
Phone: 804-864-1699
Fax: 804-864-4234

Sreerama, Sethuram
West Virginia Univ.; Dept. of Stat. & C.S.
B4, Knapp Hall
PO Box 6330
Morgantown, WV 26506
Email: sethu@cs.wvu.edu
Phone: 304-293-3607
Fax: 304-293-2272

Srivas, Mandayam
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
Email: srivas@csl.sri.com
Phone: 415-859-6136
Fax: 415-859-2844

Stoughton, Jack
Old Dominion University
ECE Dept.
Norfolk, VA 23529
Email: jws100f@eefs01.ee.odu.edu
Phone: 804-683-3741
Fax: 804-683-3220

Suri, Neeraj
Allied Signal Research Center
9140 Old Annapolis Rd.
Columbia, MD 21045
Email: suri@batc.allied.com
Phone: 410-964-4157
Fax: 410-992-5813

Tougas, Lyne
Atomic Energy Control Board
280 Slater St.
Ottawa, Ontario, CANADA KIP 559
Email: none given
Phone: 613-995-6345
Fax: 613-943-1292

Tuna, M. Esen
Derivation Systems, Inc.
5963 La Place Court
Suite 208
Carlsbad, CA 92008
Email: none given
Phone: 619-431-1400
Fax: 619-431-1484

Van Tassel, John
National Security Agency
9800 Savage Rd.
Ft. Meade, MD 20755-6000
Email: jvt@tycho.ncsc.mil
Phone: 301-688-0748
Fax: 301-688-0255

Walter, Chris J.
WW Technology Group
4519 Mustering Drum
Ellicott City, MD 21042-5949
Email: cwalter@blaze.cs.jhu.edu
Phone: 410-418-4353
Fax: 410-418-5355

Watson, James F.
NASA Langley Research Center
Mail Stop 421
Hampton, VA 23681-0001
Email: j.f.watson@larc.nasa.gov
Phone: 804-864-6985
Fax: 804-864-6327

Weinstock, Charles B.
CMU/Software Engineering Inst.
4500 Fifth Ave.
Pittsburgh, PA 15213-3890
Email: weinstoc@sei.cmu.edu
Phone: 412-268-7719
Fax: 412-268-5758

Yang, Jeffrey
The Mitre Corporation
7525 Colshire Dr.
McLean, VA 22102
Email: yang@cyclone.mitre.org
Phone: 703-883-6253
Fax: 703-883-1364

Yen, Mike
Boeing Defense & Space Group
PO Box 3707
MS 4C-63
Seattle, WA 98124-2207
Email: yen@xavier.ds.boeing.com
Phone: 206-662-0210
Fax: 206-662-0115

Young, William D.
Computational Logic, Inc.
1717 W. 6th Street
#290
Austin, TX 78703
Email: young@cli.com
Phone: 512-322-9951
Fax: 512-322-0656

Youssefi, David Hadi
Honeywell, Inc.
Air Transport Systems Div.
Box 21111
Phoenix, AZ 85036-1111
Email: none given
Phone: 602-436-6655
Fax: 602-436-5151

Zamfirescu, Alex
Intergraph Electronics
381 East Evelyn Ave.
Mountain View, CA 94041
Email: azamfirescu@ieee.org
Phone: 415-691-6426
Fax: 415-691-9016

# Appendix B:

## Comments from Attendees on the Workshop

- That was one of the best conferences I've attended. Not only were most of the talks good but there was a particularly good mix of people at the workshop who were eager to talk about their work and related ideas.
- This workshop was much more valuable than FM Europe, at a fraction of the cost.
- This was the most well organized workshop I've ever attended at Langley.
- Thanks for the conference. I picked up a lot of useful information.
- There is no regular meeting for the formal methods community. How about expanding your meetings to be yearly with an agenda set up like the GSFC SEL? Just thought I would ask.
- ... thanks again for this year's meeting. It was great fun.
- You all did a super job! Keep up the good work.
- I thoroughly enjoyed the chance to interact with people at FMWS95.
- When will the proceedings be ready? I'm looking forward to studying them.

# Appendix C:

## Detailed Overview of NASA Langley's Formal Methods Program

This paper is an expanded version of a paper presented at COMPASS 95.

# NASA Langley's Research and Technology-Transfer Program in Formal Methods

Ricky W. Butler
James L. Caldwell
Victor A. Carreño
C. Michael Holloway
Paul S. Miner

Assessment Technology Branch
NASA Langley Research Center
Hampton, Virginia

Ben L. Di Vito

VíGYAN Inc.
Hampton, Virginia

May 19, 1995

## Abstract

This paper presents an overview of NASA Langley's research program in formal methods. The major goals of this work are to make formal methods practical for use on life critical systems, and to orchestrate the transfer of this technology to U.S. industry through use of carefully designed demonstration projects. Several direct technology transfer efforts have been initiated that apply formal methods to critical subsystems of real aerospace computer systems. The research team consists of five NASA civil servants and contractors from Odyssey Research Associates, SRI International, and VíGYAN Inc.

# 1 Rationale For a Formal Methods Research Program

NASA Langley Research Center has been developing techniques for the design and validation of flight critical systems for over two decades. Although much progress has been made in developing methods to accommodate physical failures, design flaws remain a serious problem [52, 67, 35, 1, 44, 30, 74].

A 1991 report by the National Center For Advanced Technologies[1] identified "Provably Correct System Specification" and "Verification Formalism For Error-Free Specification" as key areas of research for future avionics software and ultrareliable electronics systems [2]. Aerospace engineers who attended the NASA-LaRC Flight Critical Digital Systems Technology Workshop listed techniques for the validation of concurrent and fault-tolerant computer systems high on the list of research priorities for NASA [58].

## 1.1 Why Formal Methods Is Necessary

Digital systems (both hardware and software) are notorious for their unpredictable and unreliable behavior:

> Studies have shown that for every six new large-scale software systems that are put into operation, two others are cancelled. The average software development project overshoots its schedule by half; larger projects generally do worse. And three quarters of all large systems are "operating failures" that either do not function as intended or are not used at all.
>
> Despite 50 years of progress, the software industry remains years–perhaps decades–short of the mature engineering discipline needed to meet the demands of an information-age society[31].

Lauren Ruth Wiener describes the software problem in her book, *Digital Woes: Why We Should Not Depend Upon Software*:

> Software products—even programs of modest size—are among the most complex artifacts that humans produce, and software development projects are among our most complex undertakings. They soak up however much time or money, however many people we throw at them.
>
> The results are only modestly reliable. Even after the most thorough and rigorous testing some bugs remain. We can never test all threads through the system with all possible inputs[95].

The hardware industry also faces serious difficulties, as evidenced by the recent design error in the Pentium floating-point unit. In response to an outcry over the design flaw in the Pentium floating point unit, Intel's President, Andy Grove, wrote on the comp.sys.intel Internet bulletin board:

> After almost 25 years in the microprocessor business, I have come to the conclusion that no microprocessor is ever perfect; they just come closer to perfection with each stepping. In the life of a typical microprocessor, we go thru [sic] half a dozen or more such steppings....

In a recent Washington Post article, Michael Schrage wrote:

> Pentium type problems will prove to be the rule—rather than the isolated, aberrant exceptions—as new generations of complex hardware and software hit the market. More insidious errors and harmful bugs are inevitable. That is the new reality[82].

For life critical systems, errors may mean disaster. The potential for errors is high, because these systems must not only perform their functions correctly, but also must be able to recover from the effects of failing components (in order to meet stringent ultrareliability requirements.) Often the physical fault-tolerance features of these systems are more complex and susceptible to design error than any of the basic functions of the system. John Rushby writes:

> Organization of redundancy and fault-tolerance for ultra-high reliability is a challenging problem: redundancy management can account for half the software in a flight control system and, if less than perfect can itself become the primary source of system failure [70].

[1] A technical council funded by the Aerospace Industries Association of America (AIA) that represents the major U.S. aerospace companies engaged in the research, development and manufacture of aircraft, missiles and space systems, and related propulsion, guidance, control and other equipment.

In a comprehensive assessment of formal methods [77], John Rushby discusses several notorious examples of such failures. These include the following:

- The asynchronous operation of the AFTI-F16 and sensor noise led each channel to declare the other channels failed in flight test 44. The plane was flown home on a single channel. Other potentially disastrous bugs were detected in flight tests 15 and 36.
- The HiMAT crash landed without its landing gear due to a design flaw. The problem was traced to a timing change in the software that had survived extensive testing.
- A bug in the YC-14 redundancy management was found during flight test. The bug caused a large mistracking between redundant channels.
- In flight tests of the X31, the control system went into a reversionary mode four times in the first nine flights, usually due to a disagreement between the two air data sources.
- The nationwide saturation of the AT&T switching systems on January 15, 1990 was caused by a timing problem in a fault-recovery mechanism.
- The first Shuttle mission (STS-1) was scrubbed because the fifth backup computer could not be synchronized with the other four.

Three basic strategies are advocated for dealing with the design fault problem for the life-critical system:

1. Testing (Lots of it)
2. Design Diversity (i.e. software fault tolerance: N-version programming, recovery blocks, etc.)
3. Fault Avoidance (i.e. formal specification/verification, automatic program synthesis, reusable modules)

The problem with life testing is that in order to measure ultrareliability one must test for exorbitant amounts of time. For example, to measure a $10^{-9}$ probability of failure for a 1 hour mission one must test for more than $10^9$ hours (114,000 years).

The basic idea is to use separate design and implementation teams to produce multiple versions from the same specification. At runtime, non-exact threshold voters are used to mask the effect of a design error in one of the versions. The hope is that the design flaws will manifest errors independently or nearly so. By assuming independence, one can obtain ultrareliable-level estimates of reliability, even with failure rates for the individual versions on the order of $10^{-4}$/hour. Unfortunately, the independence assumption has been rejected at the 99% confidence level in several experiments for low reliability software [47, 48].

Furthermore, the independence assumption cannot be validated for high reliability software because of the exorbitant test times required. If one cannot assume independence then one must measure correlations. This is infeasible as well; it requires as much testing time as life-testing the system, because the correlations must be in the ultrareliable region in order for the system to be ultrareliable. Therefore, it is not possible, within feasible amounts of testing time, to establish that design diversity achieves ultrareliability. Consequently, design diversity can create an "illusion" of ultrareliability without actually providing it. For a more detailed discussion, see [15, 14].

We believe that formal methods offer the only intellectually defensible method for handling design faults. Since the often quoted $1-10^{-9}$ reliability is clearly beyond the range of quantification, we have no choice but to develop life critical systems in the most rigorous manner available to us, which is use of formal methods.

## 1.2 What is Formal Methods

Engineering relies heavily on mathematical models and calculation to make judgments about designs. This is in stark contrast to the way in which software systems are designed—with *ad hoc* technique and after-implementation testing. Formal methods bring to software and hardware design the same advantages that other engineering endeavors have exploited: mathematical analysis based on models. Formal methods are used to specify and model the behavior of a system and to formally verify that the system design and implementation satisfy functional and safety properties. In principle, these techniques can produce error-free design; however, this requires a complete verification from the requirements down to the implementation, which is rarely done in practice.

Thus, formal methods is the applied mathematics of computer systems engineering. It serves a similar role in computer design as Computational Fluid Dynamics (CFD) plays in aeronautical design, providing a means of calculating and hence predicting what the behavior of a digital system will be prior to its implementation.

The tremendous scientific potential of formal methods has been recognized by theoreticians for a long time, but the formal techniques have remained the province of a few academicians, with only a few exceptions such as the Transputer [3] and the IBM CICS project [38]. The first five years of NASA Langley's program have advanced the capabilities of formal methods to the point where commercial exploitation is near.

There are many different types of formal methods with various degrees of rigor. The following is a useful (first-order) taxonomy of the degrees of rigor in formal methods:

*Level-1:* Formal specification of all or part of the system.

*Level-2:* Formal specification at two or more levels of abstraction and paper and pencil proofs that the detailed specification implies the more abstract specification.

*Level-3:* Formal proofs checked by a mechanical theorem prover.

*Level 1* represents the use of mathematical logic, or a specification language that has a formal semantics, to specify the system. This can be done at several levels of abstraction. For example, one level might enumerate the required abstract properties of the system, while another level describes an implementation that is algorithmic in style.

*Level 2* formal methods goes beyond Level 1 by developing pencil-and-paper proofs that the concrete levels logically imply the abstract, property-oriented levels. *Level 3* is the most rigorous application of formal methods. Here one uses a semi-automatic theorem prover to make sure that all of the proofs are valid. The Level 3 process of *convincing* a mechanical prover is really a process of developing an argument for an ultimate skeptic who must be shown every detail.

It is important to realize that formal methods is not an all-or-nothing approach. The application of formal methods to the most critical portions of a system is a pragmatic and useful strategy. Although a complete formal verification of a large complex system is impractical at this time, a great increase in confidence in the system can be obtained by the use of formal methods at key locations in the system. For more information on the basic principles of formal methods, see [16].

# 2 Goals of Our Program, Strategy, and Research Team

The major goals of the NASA Langley research program are to make formal methods practical for use on life critical systems developed in the United States, and to orchestrate the transfer of this technology to industry through use of carefully designed demonstration projects. Our intention is to concentrate our research efforts on the technically challenging areas of digital flight-control systems design that are currently beyond the state-of-the-art, while initiating demonstration projects in problem domains where current formal methods are adequate. The challenge of the demonstration projects should not be underestimated. That which is feasible for experts that have developed the tools and methods is often difficult for practitioners in the aerospace industry. There is often a long "learning curve" associated with the tools, the tools are not production-quality, and the tools have few or no examples for specific problem domains. Therefore, we are setting up cooperative efforts between industry and the developers of the formal methods to facilitate the technology transfer process.

This strategy leverages the huge investment of ARPA and the National Security Agency in development of tools and concentrates on the problems specific to the aerospace problem domain. NASA Langley has not sponsored the development of any general-purpose theorem provers. However, the technology transfer projects have lead to significant improvements in the Prototype Verification System (PVS) theorem prover[70] that SRI International (SRI) is developing. Several domain-specific tools are being sponsored: (1) Tablewise, (2) VHDL-analysis tool, and (3) DRS. These tools are discussed in later sections.

It is also important to realize that formal methods include a large class of mathematical techniques and tools. Methods appropriate for one problem domain may be totally inappropriate for other problem domains.

The following are some of the specific domains in which our program has concentrated: (1) architectural-level fault tolerance, (2) clock-synchronization, (3) interactive consistency, (4) design of hardware devices such as microprocessors, memory management units, DMA controllers, (5) asynchronous communication protocols, (6) design and verification of application-specific integrated circuits (ASICS), (7) Space Shuttle software, (8) navigation software, (9) decision tables, (10) railroad signaling systems.

We are also interested in applying formal methods to many different portions of the life-cycle, such as (1) requirements analysis, (2) high-level design, (3) detailed design, and (4) implementation.

Often, there is a sizable effort associated with the development of the background mathematical theories needed for a particular problem domain. Although such theories are reusable and in the long run can become "cost-effective", the initial costs can be a deterrent for industry. Therefore, one of the goals of the NASA Langley program is to build a large body of background theories needed for aerospace applications.

We also have been involved with standards activities in order to strengthen the United States commitment to safety.

## 2.1 Technology Transfer

The key to successful technology transfer is building a cooperative partnership with a customer. In order for this partnership to work, NASA Langley must become directly involved in specific problem domains of the aerospace industry[2]. NASA must also effectively communicate its basic research accomplishments in a manner that reveals a significant potential benefit to the aerospace community. Equally important is the need for industry to make an investment to work together with NASA on joint projects to devise demonstration projects that are realistic and practical. The ultimate goal of our technology transfer process is for formal methods to become the "state-of-the-practice" for U.S. industry development of ultrareliable digital avionics systems. However, before we can develop new tools and techniques suitable for adoption by industry, we must work with the system developers in industry to understand their needs. We must also overcome the natural skepticism that industry has of any new technology.

Our basic approach to technology transfer is as follows. The first step is to find an industry representative who has become interested in formal methods, believes that there is a potential benefit of such methods, and is willing to work with us. The next step is to fund our formal methods research team to apply formal methods to an appropriate example application. This process allows the industry representative to see what formal methods are and what they have to offer, and it allows us (the formal methods team) to learn the design and implementation details of state-of-the-practice components so we can better tailor our tools and techniques to industry's needs. If the demonstration project reveals a significant potential benefit, the next stage of the technology transfer process is for the industry representative to initiate an internal formal methods program, and begin a true cooperative partnership with us.

Another important part of our technology transfer strategy is working with the Federal Aviation Administration (FAA) to update certification technology with respect to formal methods. If the certification process can be redefined in a manner that awards credit for the use of formal methods, a significant step towards the transfer of this technology to the commercial aircraft industry will have been accomplished.

Langley has also been sponsoring a series of workshops on formal methods. The first workshop, held in August 1990, focused on building cooperation and communication between U.S. formal methods researchers[18]. The second, held in August 1992, focused on education of the U.S. aerospace industry about formal methods[43]. A third workshop will be held in May 1995.

Another component of our technology transfer strategy, is to use the NASA's Small Business Innovative Research (SBIR) program to assist small businesses in the development of commercially viable formal methods tools and techniques. The first contracts under the program began in early 1994.

Finally, to facilitate technology transfer, much information on NASA Langley's formal methods research is available on the Internet via either anonymous FTP or World Wide Web. PostScript and DVI versions of many research papers are available through anonymous FTP on machine `deduction.larc.nasa.gov` (IP address: `128.155.18.16`) in directory `pub/fm`. This directory, and much more information, is also available through World Wide Web, using the following Uniform Resource Locator:

[2] To date, our efforts have concentrated on the aerospace industry, but we are actively seeking partners from other industries also.

http://atb-www.larc.nasa.gov/fm.html

## 2.2 FAA/RTCA Involvement

As the federal agency responsible for certification of civil air transports, the FAA shares our interest in promising approaches to engineering and validating ultrareliable flight-control systems. Additionally, because the FAA must approve any new methodologies for developing life-critical digital systems for civil air transports, their acceptance of formal methods is a necessary precursor to its adoption by industry system designers. We are working with Pete Saraceni of the FAA Technical Center and Mike DeWalt, FAA National Resource Specialist for Software, to insure that our program is relevant to the certification process. The FAA has co-sponsored some of our work. John Rushby of SRI gave a tutorial on formal methods at an FAA Software Advisory Team (SWAT) meeting at their request. The SWAT team suggested that we include an assessment of formal methods in an ongoing Guidance Control Software (GCS) experiment in our branch; Odyssey Research Associates (ORA) developed a formal specification of the GCS application.

John Rushby has written a chapter for the FAA Digital Systems Validation Handbook Volume III on formal methods[20]. The handbook provides detailed information about digital system design and validation and is used by the FAA certifiers. In preparation for this chapter, Rushby produced a comprehensive analysis of formal methods [77].

George Finelli, the former assistant Branch Head of the System Validation Methods Branch (the Branch in which the formal methods team worked before NASA Langley's reorganization in 1994) and a member of the RTCA committee formed to develop DO–178B, together with Ben Di Vito (ViGYAN Inc.), was instrumental in including formal methods as an alternate means of compliance in the DO–178B standard.

Currently, members of the Langley staff are involved in RTCA committees SC-180 (Airborne Electronic Hardware) and SC-182 (Minimal Operating Performance Standard for an Airborne Computer Resource).

## 2.3 Team

The Langley formal methods program involves both local researchers and industrial / academic researchers working under contract to NASA Langley. Currently the local team consists of five civil servants and one contractor (ViGYAN Inc.). The lead NASA Langley formal methods researcher, Ricky W. Butler, may be contacted through electronic mail to `R.W.Butler@LaRC.NASA.GOV`.

NASA Langley has recently awarded two five-year task-assignment contracts specifically devoted to formal methods (from the competitive NASA RFP 1-132-DIC.1021). The selected contractors were SRI International (SRI) and Odyssey Research Associates (ORA). This was a follow-on contract from the previous competitive contract that had awarded three contracts to SRI, ORA, and Computational Logic Inc. (CLI).

# 3 Current Technology Development and Transfer Projects

## 3.1 AAMP5/AAMP-FV Project

In 1993, NASA Langley initiated a joint project involving Collins Commercial Avionics and SRI International. The goal was to investigate the application of formal techniques to a commercial microprocessor design, the Collins AAMP5 microprocessor. The AAMP5 is the latest member of the CAPS/AAMP family of microprocessors and is object code compatible with the AAMP2 processor [4]. The CAPS/AAMP family of microprocessors has been widely used by the commercial and military aerospace industries. Some examples of use of earlier members of the family include:

1. Boeing 747-400 Integrated Display System (IDS)
2. Boeing 737-300 Electronic Flight Instrumentation System (EFIS)
3. Boeing 777 Flight Control Backdrive
4. Boeing 757,767 Autopilot Flight Director System (AFDS)

5. military and commercial Global Positioning (GPS) Systems.

The first phase of the project consisted of the formal specification of the AAMP5 instruction set and microarchitecture using SRI's PVS [90, 91, 11, 69, 68, 87]. While formally specifying the microprocessor, two design errors were discovered in the microcode. These errors were uncovered as a result of questions raised by the formal methods researchers at Collins and SRI while seeking to formally specify the behavior of the microprocessor[59]. The Collins formal methods team believes that this effort has prevented two significant errors from going into the first fabrication of the microprocessor.

The second phase of the project consisted of formally verifying the microcode of a representative subset of the AAMP5 instructions. Collins seeded two errors in the microcode provided to SRI in an attempt to assess the effectiveness of formal verification. Both of these errors (and suggested corrections) were discovered while proving the microcode correct[59]. It is noteworthy that both the level 2 and level 3 applications of formal methods were successful in finding bugs. Based on the success of the AAMP5 project, a new effort has been initiated with Rockwell-Collins to apply formal methods in the design level verification of a microprocessor, currently designated as AAMP-FV.

## 3.2 Tablewise Project

Under NASA funding, Odyssey Research Associates is working with Honeywell Air Transport Systems Division (Phoenix) to study the incorporation of formal methods into the company's software development processes. Because Honeywell uses decision tables to specify the requirements and designs for much of their software[3], ORA is developing a prototype tool, called Tablewise[4], to analyze the characteristics of decision tables. Tablewise uses a generalization of Binary Decision Diagrams to determine if a particular table is exclusive (for every combination of parameter values, at most one action can be chosen) and exhaustive (for every combination of parameter values, at least one action can be chosen). The tool is also capable of automatically generating documentation and Ada code from a decision table [37]. We consider this a level 3 application of formal methods: although a general purpose prover is not used, the analysis is mechanized in a computer program.

In 1995, ORA will develop algorithms to handle advanced analysis of decision tables. Two particular areas of analysis that will be considered are testing of additional properties of tables and techniques for efficiently handling partitioned tables. The Honeywell personnel involved in the project hope that the concepts developed in the Tablewise project can be incorporated into an industrial-strength tool that will significantly reduce the effort required to develop new software.

## 3.3 Union Switch and Signal

As part of a joint research agreement, NASA Langley formal methods researchers are collaborating with engineers at Union Switch and Signal (US&S) to use formal methods in the design of railway switching and control applications. Railway switching control systems, like digital flight control systems, are safety critical systems. US&S is the leading U.S. supplier of railway switching control systems. Their Advanced Technology Group, lead by Dr. Joseph Profeta, has applied formal methods in past efforts and turned to NASA for expertise in integrating these techniques into their next generation products.

The initial project, started in 1993, was a cooperative effort between NASA, US&S, and Odyssey Research Associates. The result of this first year's work was a formal mathematical model of a railway switching network, defined in two levels. The top level of the model provides the mechanisms for defining the basic concepts: track, switches, trains and their positions and control liners of a train (i.e. how far down the track it has clearance to travel.) The second level is a formalization of the standard scheme used in railroad control, the block model control system. A level 2 proof that the fixed block control system is "safe" with respect to the top level model has been completed. Models of US&S proprietary control schemes were also formulated.

[3] A decision table is a tabular format for defining the rules that choose a particular action to perform based on the values of certain parameters.

[4] previously called Tbell.

The European formal methods community has addressed safety properties of certain components of railroad control systems, but the work there has typically been at lower levels. The cooperative work with US&S is unique in that a high level model of a railroad system has been described and used to analyze the safety of various control schemes.

The next phase of the collaborative effort will concentrate on formal modeling and analysis of the fault-tolerant core of US&S's next generation fail-stop control architecture.

## 3.4 Space Applications

A team spread across three NASA centers has been formed to study the application and technology transfer of formal methods to NASA space programs. A consortium of researchers and practitioners from LaRC, JSC, and JPL, together with support from Loral Space Information Systems, SRI International, and VíGYAN Inc., has been actively pursuing this objective since late 1992. The near term goal is to define and carry out pilot projects using portions of existing large-scale space programs. The long term goal is to enable organizations such as Loral to reduce formal methods to practice on programs of national importance.

The NASA Formal Methods Demonstration Project for Space Applications focuses on the use of formal methods for requirements analysis because the team believes that formal methods are more practically applied to requirements analysis than to late-lifecycle development phases [40]. A series of trial projects was conducted and cost effectiveness data were collected. The team's efforts in 1993 were concentrated on a single pilot project, while later efforts in 1994 have been more diffuse.

The 1993 project was the formal specification of a very mature piece of the Space Shuttle flight control requirements called Jet Select. Initial specifications were written to capture an existing, low-level statement of the requirements. Few proofs were produced for the first specification, but 46 issues were identified and several minor errors were found in the requirements. A second specification was produced for an abstract (i.e., high level) representation of the Jet Select requirements. This abstraction, along with the 24 proofs of key properties, was accomplished in under 2 work months, and although it only uncovered 6 issues, several of these issues were significant.

NASA Langley's primary role in 1994 included support for two Space Shuttle software change requests (CR). One CR concerns the integration of new Global Positioning System (GPS) functions while the other concerns a new function to control contingency aborts known as Three Engine Out (3 E/O). Both of these tasks involve close cooperation among formal methods researchers at NASA Langley, VíGYAN Inc., and SRI International with requirements analysts from Loral Space Information Systems.

The Space Shuttle is to be retrofitted with GPS receivers in anticipation of the TACAN navigation system being phased out by the DoD. Additional navigation software will be incorporated to process the position and velocity vectors generated by these receivers. A decision was made to focus the trial formal methods task on just a few key areas because the CR itself is very large and complex. A set of preliminary formal specifications was developed for the new Shuttle navigation principal functions known as GPS Receiver State Processing and GPS Reference State Processing, using the language of SRI's Prototype Verification System (PVS). While writing the formal specifications, 43 minor discrepancies were detected in the CR and these have been reported to Loral requirements analysts.

The Three Engine Out (3 E/O) Task is executed each cycle during powered flight until either a contingency abort maneuver is required or progress along the powered flight trajectory is sufficient to preclude a contingency abort even if three main engines fail. The 3 E/O task consists of two parts: 3 E/O Region Selection and 3 E/O Guidance. 3 E/O Region Selection is responsible for selecting the type of external tank (ET) separation maneuver and assigning the corresponding region index. 3 E/O guidance monitors ascent parameters and determines if an abort maneuver is necessary.

We have developed and analyzed a formal model of the series of sequential maneuvers that comprise the 3 E/O algorithm. To date, 20 potential issues have been found, including undocumented assumptions, logical errors, and inconsistent and imprecise terminology. These findings are listed as potential issues pending review by the 3 E/O requirements analyst.

The GPS and 3 E/O tasks has continued into 1995. We hope to get formal methods incorporated as a requirements analysis technique for Space Shuttle software. In addition, NASA Langley contributed to a NASA guidebook under development by the inter-center team. The first volume of the guidebook is intended for managers of NASA projects who will be using formal methods in requirements analysis activities. A second volume is planned that will be aimed at practitioners. NASA will publish the first volume early in 1995, with the second volume expected by early 1996.

### 3.5 Requirements Analysis

Better methods for writing and analyzing requirements is one of the greatest needs that commercial industry faces today. Requirements are usually incomplete, poorly defined, and change rapidly as a system is developed. Errors introduced in requirements are often the most serious because they manifest themselves as major design errors and are often very expensive to correct when they are discovered, usually late in the life-cycle during implementation testing.

NASA Langley is sponsoring work to develop special interfaces to PVS, a formal specification language and theorem proving environment developed by SRI. The goal of this work is to develop an interface that (1) is readable by Domain.Experts and typical customers, (2) is precise enough to support formal analysis, (3) supports concurrent development by many individuals, and (4) discourages overspecification.

### 3.6 NASA Small Business Innovative Research Program

In 1993, a formal methods subtopic was a part of the NASA Small Business Innovative Research (SBIR) solicitation. Two proposals were selected for 6-month Phase I funding for 1994: *VHDL Lightweight Tools*, by Odyssey Research Associates, and *DRS — Derivation Reasoning System, A Digital Design Derivation System for Hardware Synthesis*, by Derivation Systems, Inc. of Bloomington, Indiana. After the completion of the Phase I efforts, Derivation Systems' proposal for Phase II funding was accepted, and contract negotiations are currently underway to initiate a 2-year effort. Contracts for these efforts just recently began.

## 4 Past Efforts

This section describes previous work in each of the following four focus areas: fault-tolerant systems, verification of software, verification of hardware devices, and civil air transport requirements specification.

### 4.1 Fault-tolerant Systems

The goal of this focus area was to create a formalized theory of fault tolerance including redundancy management, clock synchronization, Byzantine agreement, voting, etc. Much of the theory developed here is applicable to future fault-tolerant systems designs. A detailed design of a fault-tolerant reliable computing base, the Reliable Computing Platform (RCP), has been developed and proven correct. It is hoped that the RCP will serve as a demonstration of the formal methods process and provide a foundation that can be expanded and used for future aerospace applications. It is one of the largest formal verifications ever performed.

The RCP architecture was designed in accordance with a system-design philosophy called "Design For Validation" [42, 41]. The basic tenets of this design philosophy are as follows:

1. A system is designed in such a manner that complete and accurate models can be constructed to estimate critical properties such as reliability and performance. All parameters of the model that cannot be deduced from the logical design must be measured. All such parameters must be measurable within a feasible amount of time.

2. The design process makes tradeoffs in favor of designs that minimize the number of parameters that must be measured in order to reduce the validation cost. A design that has exceptional performance properties yet requires the measurement of hundreds of parameters (for example, by time-consuming fault-injection experiments) would be rejected over a less capable system that requires minimal experimentation.

3. The system is designed and verified using rigorous mathematical techniques, usually referred to as a formal verification. It is assumed that the formal verification makes system failure due to design faults negligible so the reliability model does not include transitions representing design errors.

4. The reliability (or performance) model is shown to be accurate with respect to the system implementation. This is accomplished analytically not experimentally.

Thus, a major objective of this approach is to minimize the amount of experimental testing required and maximize the ability to reason mathematically about correctness of the design. Although testing cannot be eliminated from the design/validation process, *the primary basis of belief in the dependability of the system must come from analysis rather than from testing.*

### 4.1.1 The Reliable Computing Platform

The Reliable Computing Platform dispatches control-law application tasks and executes them on redundant processors as illustrated in figure 1. The intended applications are safety critical with reliability requirements on the order of $1 - 10^{-9}$. The reliable computing platform performs the necessary fault-tolerant functions and provides an interface to the network of sensors and actuators.

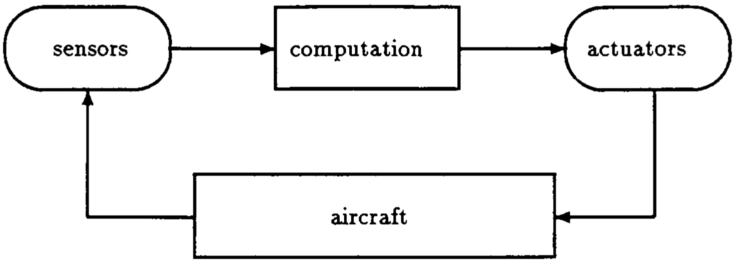


Figure 1: Intended Application of RCP

The RCP operating system provides the applications software developer with a reliable mechanism for dispatching periodic tasks on a fault-tolerant computing base that *appears* to him as a single ultrareliable processor. A multi-level hierarchical specification of the RCP is shown in figure 2.

The top level of the hierarchy describes the operating system as a function that sequentially invokes application tasks. This view of the operating system will be referred to as the *uniprocessor specification (US)*, which is formalized as a state transition system and forms the basis of the specification for the RCP. Fault tolerance is achieved by voting results computed by the replicated processors operating on the same inputs. Interactive consistency checks on sensor inputs and voting of actuator outputs require synchronization of the replicated processors. The second level in the hierarchy (RS) describes the operating system as a synchronous system where each replicated processor executes the same application tasks. The existence of a global time base, an interactive consistency mechanism and a reliable voting mechanism are assumed at this level.

Level 3 of the hierarchy (DS) breaks a frame into four sequential phases. This allows a more explicit modeling of interprocessor communication and the time phasing of computation, communication, and voting. At the fourth level (DA), the assumptions of the synchronous model must be discharged. Rushby and von Henke [79] report on the formal verification of Lamport and Melliar-Smith's [50] interactive-convergence clock synchronization algorithm. This algorithm can serve as a foundation for the implementation of the replicated system by bounding the amount of asynchrony in the system so that it can duplicate the functionality of the DS model. Dedicated hardware implementations of the clock synchronization function are a long-term goal.

In the LE model, a more detailed specification of the activities on a local processor are presented. In particular, three areas of activity are elaborated in detail: (1) task dispatching and execution, (2) minimal voting, and (3) interprocessor communication via mailboxes. An intermediate model, DA_minv, that simplified the construction of the LE model was used. Some of the refinements occur in the DA_minv model and some in the LE model. For example, the concept of minimal voting is addressed in considerable detail

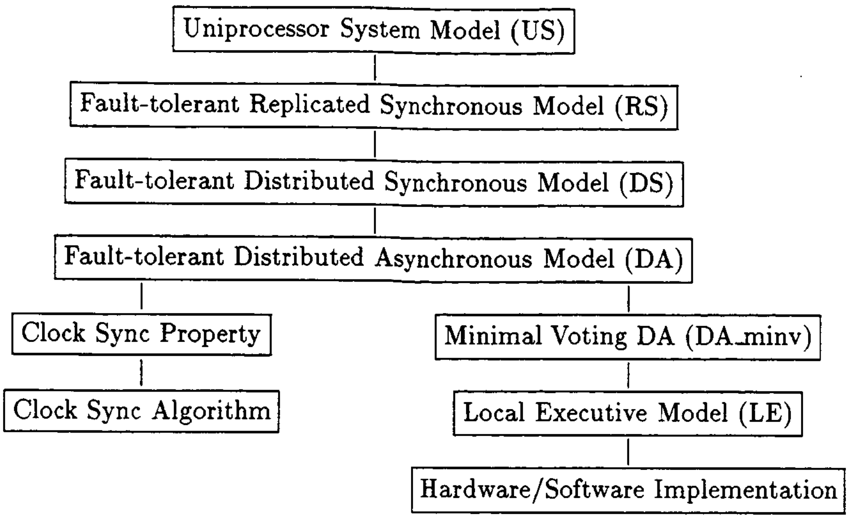

Uniprocessor System Model (US)
Fault-tolerant Replicated Synchronous Model (RS)
Fault-tolerant Distributed Synchronous Model (DS)
Fault-tolerant Distributed Asynchronous Model (DA)
Clock Sync Property
Minimal Voting DA (DA_minv)
Clock Sync Algorithm
Local Executive Model (LE)
Hardware/Software Implementation

Figure 2: Hierarchical Specification of the Reliable Computing Platform.

in the DA_minv model. Of primary importance in the LE specification is the use of a memory management unit by the local executive in order to prevent the overwriting of incorrect memory locations while recovering from the effects of a transient fault.

Figure 3 depicts the generic hardware architecture assumed for implementing the replicated system. The hardware architecture is a classic N-modular redundant (NMR) system with a small number, $N$, of processors. Single-source sensor inputs are distributed by special purpose hardware executing a Byzantine agreement algorithm. Replicated actuator outputs are all delivered in parallel to the actuators, where force-sum voting occurs. Interprocessor communication links allow replicated processors to exchange and vote on the results of task computations.

The top two levels of the RCP were originally formally specified in standard mathematical notation and connected via mathematical (i.e. level 2 formal methods) proof [25, 24, 22]. Under the assumption that a majority of processors is working in each frame, the proof establishes that the replicated system computes the same results as a single processor system not subject to failures. Sufficient conditions were developed that guarantee that the replicated system recovers from transient faults within a bounded amount of time. SRI subsequently generalized the models and constructed a mechanical proof in EHDM [75]. Next, the local team developed the third and fourth level models. The top two levels and the two new models (i.e. DS and DA) were then specified in EHDM and all of the proofs were done mechanically using the EHDM 5.2 prover [12, 23].

Both the DA_minv model and the LE model were specified formally and have been verified using the EHDM verification system[13]. All RCP specifications and proofs are available electronically via the Internet using anonymous FTP or World Wide Web (WWW) access. Anonymous FTP access is available through the host `deduction.larc.nasa.gov` using the path `pub/fm/larc/RCP-specs`. WWW access to the FTP directory is provided through the NASA Langley Formal Methods Program home page: `http://atb-www.larc.nasa.gov/fm-top.html`

### 4.1.2 Clock Synchronization

The redundancy management strategies of virtually all fault-tolerant systems depend on some form of voting, which in turn depends on synchronization. Although in many systems the clock synchronization function has not been decoupled from the applications (e.g. the redundant versions of the applications synchronize by messages), research and experience have led us to believe that solving the synchronization problem independently from the applications design can provide significant simplification of the system [49, 32]. The operating system is built on top of this clock-synchronization foundation. Of course, the correctness of

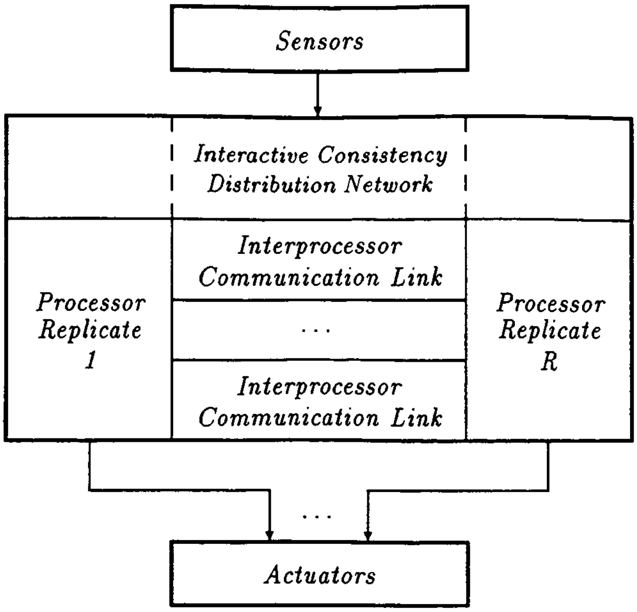

Sensors
Interactive Consistency Distribution Network
Processor Replicate 1
Interprocessor Communication Link
...
Interprocessor Communication Link
Processor Replicate R
...
Actuators

Figure 3: Generic hardware architecture.

this foundation is essential. Thus, the clock synchronization algorithm and its implementation are prime candidates for formal methods. The verification strategy shown in figure 4 is being explored.

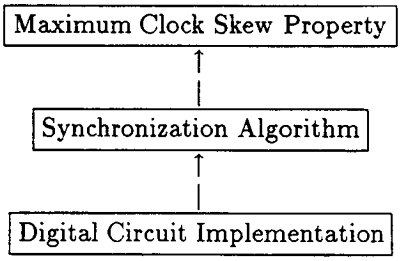

Maximum Clock Skew Property
Synchronization Algorithm
Digital Circuit Implementation

Figure 4: Hierarchical Verification of Clock Synchronization

The top-level in the hierarchy is an abstract property of the form:

$$\forall \text{ non-faulty } p, q : |C_p(t) - C_q(t)| < \delta$$

where $\delta$ is the maximum clock skew guaranteed by the algorithm as long as a sufficient number of clocks (and the processors they are attached to) are working. The function $C_p(t)$ gives the value of clock $p$ at real time $t$. The middle level in the hierarchy is a mathematical definition of the synchronization algorithm. The bottom level is a detailed digital design of a circuit that implements the algorithm. The bottom level is sufficiently detailed to make translation into silicon straight forward.

The verification process involves two important steps: (1) verification that the algorithm satisfies the maximum skew property and (2) verification that the digital circuitry correctly implements the algorithm. The first step was completed by SRI International. The first such proof was accomplished during the design and verification of SIFT [50]. The proof was done by hand in the style of journal proofs. More recently this proof step was mechanically verified using the EHDM theorem prover[79, 80]. In addition, SRI mechanically verified Schneider's clock synchronization paradigm [81] using EHDM[88, 89]. A further generalization was found at NASA Langley [60][5]. The design of a digital circuit to distribute clock values in support of fault-tolerant synchronization was completed by SRI and was partially verified.[6] CLI reproduced the SRI verification of the interactive convergence algorithm using the Boyer-Moore theorem prover [100].

NASA Langley researchers designed and implemented a fault-tolerant clock synchronization circuit capable of recovery from transient faults [62, 61, 60]. The top-level specification for the design is the EHDM verification of Schneider's paradigm. The circuit was implemented with programmable logic devices (PLDs) and FOXI fiber optic communications chips [63].

Using a combination of formal techniques, a verified clock synchronization circuit design has also been developed[64]. The principal design tool was the Digital Design Derivation system (DDD) developed by Indiana University[8]. Some design optimizations that were not possible within DDD were verified using PVS.

### 4.1.3 Byzantine Agreement Algorithms

Fault-tolerant systems, although internally redundant, must deal with single-source information from the external world. For example, a flight control system is built around the notion of feedback from physical sensors such as accelerometers, position sensors, and pressure sensors. Although these can be replicated (and they usually are), the replicates do not produce identical results. To use bit-by-bit majority voting, all of the computational replicates must operate on identical input data. Thus, the sensor values (the complete redundant suite) must be distributed to each processor in a manner which guarantees that all working processors receive exactly the same value even in the presence of some faulty processors. This is the classic Byzantine Generals problem [51]; algorithms to solve the problem are called Byzantine agreement algorithms. CLI investigated the formal verification and implementation of such algorithms. They formally verified the original Marshall, Shostak, and Lamport version of this algorithm using the Boyer Moore theorem prover [5]. They also implemented this algorithm down to the register-transfer level and demonstrated that it implements the mathematical algorithm [6], and then subsequently verified the design down to a hardware description language HDL developed at CLI [66]. A more efficient mechanical proof of the oral messages algorithm was also developed by SRI[76].

ORA also investigated the formal verification of Byzantine Generals algorithms. They focused on the practical implementation of a Byzantine-resilient communications mechanism between Mini-Cayuga microprocessors [92, 7]. The Mini-Cayuga is a small but formally verified microprocessor developed by ORA. It is a research prototype and has not been fabricated. The communications circuitry would serve as a foundation for a fault-tolerant architecture. It was designed assuming that the underlying processors were synchronized (say by a clock synchronization circuit). The issues involved with connecting the Byzantine communications circuit with a clock synchronization circuit and verifying the combination has not yet been explored.

Thambidurai and Park [94] introduced a fault model that classified faults into three categories: asymmetric, symmetric, and benign. They further suggested the need for and developed an algorithm that had capabilities beyond that of the earlier Byzantine generals algorithms. In particular, their algorithm can mask the effects of a less severe class of faults, in a more effective way. SRI has formally verified an improved version of this algorithm [55, 54, 56]

The newly developed hybrid-fault theory was then applied to the analysis of the Charles Stark Draper Labs "Fault-Tolerant Processor" (FTP). A unique feature of this architecture is its use of "interstages" to relay messages between processors. These are significantly smaller than a processor and lead to an asymmetric architecture that is far more efficient than the traditional Byzantine agreement architectures. The SRI work not only formalized the existing informal analysis but extended it to cover a wider range of faulty behavior[57].

Also SRI subsequently generalized their clock synchronization work to encompass the hybrid fault model [78].

## 4.2 Verification of Software

Our past software verification projects are described in this section.

[5] The bounded delay assumption was shown to follow from the other assumptions of the theory.

[6] Unlike the NASA circuit, the SRI intent is that the convergence algorithm be implemented in software.

### 4.2.1 Formal Specification of Space Shuttle Jet Select

NASA Langley worked with NASA Johnson Space Center and the Jet Propulsion Laboratory (JPL) in a study to explore the feasibility and utility of applying mechanically-supported formal methods to requirements analysis for space applications. The team worked jointly to develop a formal specification of the Jet Select function of the NASA Space Shuttle, which is a portion of the Shuttle's Orbit Digital Auto-Pilot (DAP). Specifications were written at three different levels of abstraction. The highest level specifications were proved to meet a set of critical properties. This formal analysis uncovered hidden problems in a highly critical and mature FSSR specification for Shuttle. This project impressed several key members of the Shuttle software community that the benefits of formal methods are concrete and economically realizable. A very favorable reaction was received from the IBM (now Loral) requirements analysts and senior JSC personnel (Bob Hinson, in particular). They would like to work with us "to build a different paradigm where engineers write requirements like this before passing the requirements to software development."

This demonstration project was funded by the Office of Safety and Mission Quality at NASA Headquarters, which controls funding for verification and validation of all major NASA space projects.

### 4.2.2 Honeywell Navigation Specification

A cooperative research effort was initiated in 1993 with Honeywell Air Transport Systems Division (Phoenix) to study the incorporation of formal methods into the company's software development processes. In the initial project in this effort, NASA Langley funded ORA to identify a component of the Boeing 777 system to which formal specification techniques could be applied, and to develop the formal specifications for that component. ORA, in collaboration with personnel from Langley and Honeywell, chose the navigation subsystem as a suitable application.

Using documents supplied to them by Honeywell, ORA developed a specification that addressed the following aspects of navigation:

- basic mathematical concepts such as functions over the reals, and physical units such as distance, velocity, and acceleration
- definition of objects such as aircraft, radios, sensors, navigation aids, and the navigation database
- definition of algorithms such as complementary filter processing, navigation aid selection, navigation mode selection, and position determination
- relating the mathematical model to Ada by partitioning the system in Ada package specifications, and annotating individual Ada functions and procedures with formal specifications

The specification was done using ORA's Penelope tool.

### 4.2.3 Verification of Existing Ada Applications Software

Odyssey Research Associates completed two tasks applying their Ada verification tools to aerospace applications. The first task was to verify some utility routines obtained from the NASA Goddard Space Flight Center and the NASA Lewis Research Center using their Ada Verification Tool named Penelope [33]. This task was accomplished in two steps: (1) formal specification of the routines and (2) formal verification of the routines. Both steps uncovered errors [26]. The second task was to formally specify the mode-control panel logic of a Boeing 737 experimental aircraft system using Larch (the specification language used by Penelope) [34].

## 4.3 Verification of Hardware Devices

Our past research and technology transfer efforts in the area of formal verification of hardware devices are described below.

### 4.3.1 Boeing Hardware Devices

The Boeing Company was contracted by NASA Langley to develop advanced validation and verification techniques for fly-by-wire systems. As part of the project, Boeing explored the use of formal methods. The goal of this work was two-fold: (1) technology transfer of formal methods to Boeing, and (2) assessment of formal methods technology maturity.

The first phase of this project focused on the formal verification of "real" hardware devices using the HOL hardware verification methodology. With the assistance of a subcontract with U. C. Davis, Boeing partially verified a set of hardware devices, including a microprocessor[98], a floating-point coprocessor similar to the Intel 8087 but smaller[72, 71], a direct memory access (DMA) controller similar to the Intel 8237A but smaller[46], and a set of memory-management units[86, 83]. U. C. Davis also developed the generic-interpreter theory to aid in the formal specification and verification of hardware devices[99, 97, 96], and a horizontal-integration theory for composing verified devices into a system[85, 84, 73, 45]. After demonstrating the feasibility of verifying standard hardware devices, Boeing applied the methodology to a proprietary hardware device called the Processor Interface Unit (PIU) that is being developed for aeronautics and space applications[29].

Boeing and U.C. Davis also performed an assessment of the U.K. Royal Signals and Radar Establishment's (RSRE) VIPER chip [53]. This was part of a now-completed 3 year Memorandum of Understanding (MOU) with RSRE. CLI and Langley researchers also performed assessments of the VIPER project[10, 19, 17].

Application of formal methods to the suite of Intel-like devices and the PIU demonstrated that formal methods can be practically applied to the digital hardware devices being developed by Boeing today and provided insight on how to make the process more cost effective.

### 4.3.2 CSDL Scoreboard Hardware

A joint project between ORA and Charles Stark Draper Laboratory (CSDL) was completed in 1993. NASA Langley and the Army had been funding CSDL to build fault-tolerant computer systems for over two decades. During this project, CSDL became interested in the use of formal methods to increase confidence in their designs. ORA was given the task of formally specifying and verifying a key circuit (called the scoreboard) of the Fault-Tolerant Parallel Processor (FTPP) [36] in Clio [93]. The formal verification uncovered previously unknown design errors. When the scoreboard chip was fabricated, it worked without any error manifestation. It was the first time that CSDL produced a chip that worked "perfectly" on a first fabrication. CSDL credits VHDL-development tools and formal methods for the success.

### 4.3.3 Asynchronous Communication

CLI developed a formal model of asynchronous communication and demonstrated its utility by formally verifying a widely used protocol for asynchronous communication called the bi-phase mark protocol, also known as "Bi-Φ-M," "FM" or "single density" [65]. It is one of several protocols implemented by microcontrollers such as the Intel 82530 and is used in the Intel 82C501AD Ethernet Serial Interface.

### 4.3.4 Digital Design Derivation

Funded in part by a NASA Langley Graduate Student Research Program fellowship, Bhaskar Bose developed the Digital Design Derivation system (DDD) and used it to design a verified microprocessor. DDD implements a formal design algebra that allows a designer to transform a formal specification into a correct implementation[8]. Bose formally derived the DDD-FM9001[9] microprocessor from Hunt's top-level specification of the FM9001 microprocessor[39].

## 4.4 Civil Air Transport Requirements Specification

Work with Boeing to develop a prototype interface for formal requirements analysis of a civil air transport was completed in 1992[27, 28]. This work, performed under a subcontract to California Polytechnic State University, included development of a Wide-Spectrum Requirements Specification Language (WSRSL) and prototype tools to support the language. Portions of a set of requirements for an Advanced Subsonic Civil

Transport (ASCT) developed by a Boeing engineer under previous NASA funding were rewritten in WSRSL to demonstrate the use of the language and toolset. Since WSRSL is a formal language, the specifications can be formally analyzed for syntactic correctness, completeness, and consistency.

## 5 Summary

The NASA Langley program in formal methods has two major goals: (1) develop formal methods technology suitable for a wide range of aerospace designs and (2) facilitate technology transfer by initiating joint projects between formal methods researchers and aerospace industries to apply the results of the research to real systems. Starting in 1991, NASA Langley initiated several aggressive projects designed to move FM into productive use in the aerospace community:

- Boeing PIU Project (1991)
- Charles Stark Draper FTPP Scoreboard Project (1991)
- Allied Signal Hybrid Fault Models (1992)
- Shuttle Tile Project (1992)
- Space Shuttle Jet Select Project (1993)[7]
- Honeywell Navigation (1993)
- Rockwell Collins AAMP5 (1993)
- Honeywell Tablewise (1994)
- Union Switch and Signal (1994)
- Rockwell Collins AAMP-FV (1995)

NASA's program has advanced aerospace-related formal methods in the United States to the point where commercial exploitation of formal methods is near. Our program has driven the development of PVS, the most advanced general-purpose theorem prover in the world [70], and the Odyssey Research Associates VHDL-verification tool. Commercial industry has been anxious to work with our team, although we have not had sufficient resources to work with as many as we would have liked. Nevertheless, we have helped lay the necessary foundation for productive use of formal methods in several companies.

Fundamental research has been performed in the design and verification of a fault-tolerant reliable computing platform that can support real-time control applications. Also much progress has been made in developing and demonstrating formal methods for critical subsystems of the RCP such as clock synchronization, Byzantine agreement, and voting.

## References

[1] Saab Blames Gripen Crash on Software. *Aviation Week & Space Technology*, Feb. 1989.

[2] *Key Technologies For the Year 2000.* National Center for Advanced Technologies, 1250 Eye Street N.W., Washington, D.C. 20005, June 1991.

[3] Barrett, Geoff: Formal Methods Applied to a Floating-Point Number System. *IEEE Transactions on Software Engineering*, vol. 15, no. 5, May 1989, pp. 611–621.

[4] Best, David W.; Charles E. Kress, Nick M. Mykris; Russell, Jeffrey D.; and Smith, William J.: An advanced-architecture CMOS/SOS microprocessor. *IEEE Micro*, vol. 2, no. 4, Aug. 1982, pp. 11–26.

[7] This project was not initiated by Langley, but Langley has been a major participant in it.

[5] Bevier, William R.; and Young, William D.: *Machine Checked Proofs of the Design and Implementation of a Fault-Tolerant Circuit.* NASA Contractor Report 182099, Nov. 1990.

[6] Bevier, William R.; and Young, William D.: The Proof of Correctness of a Fault-Tolerant Circuit Design. In *Second IFIP Conference on Dependable Computing For Critical Applications*, Tucson, Arizona, Feb. 1991, pp. 107–114.

[7] Bickford, Mark; and Srivas, Mandayam: *Verification of the FtCayuga Fault-Tolerant Microprocessor System (Volume 2: Formal Specification and Correctness Theorems).* NASA Contractor Report 187574, July 1991.

[8] Bose, Bhaskar: *DDD - A Transformation System for Digital Design Derivation.* Indiana University, Technical Report 331, Computer Science Department, May 1991.

[9] Bose, Bhaskar; and Johnson, Steven D.: DDD-FM9001: Derivation of a Verified Microprocessor. An Exercise in Integrating Verification with Formal Derivation. In Milne, G.; and Pierre, L., editors 1993:, *Proceedings of IFIP Conference on Correct Hardware Design and Verification Methods.* Springer, LNCS 683, 1993, pp. 191–202. also published as Tech Report # 380, Computer Science Department, Indiana University.

[10] Brock, Bishop; and Hunt, Jr., Warren A.: *Report on the Formal Specification and Partial Verification of the VIPER Microprocessor.* NASA Contractor Report 187540, July 1991.

[11] Butler, Ricky W.: *An Elementary Tutorial on Formal Specification and Verification Using PVS.* NASA Technical Memorandum 108991, Sept. 1993.

[12] Butler, Ricky W.; and Di Vito, Ben L.: *Formal Design and Verification of a Reliable Computing Platform For Real-Time Control (Phase 2 Results).* NASA Technical Memorandum 104196, Jan. 1992.

[13] Butler, Ricky W.; Di Vito, Ben L.; and Holloway, C. Michael: *Formal Design and Verification of a Reliable Computing Platform For Real-Time Control (Phase 3 Results).* NASA Technical Memorandum 109140, Aug. 1994.

[14] Butler, Ricky W.; and Finelli, George B.: The Infeasibility of Experimental Quantification of Life-Critical Software Reliability. In *Proceedings of the ACM SIGSOFT '91 Conference on Software for Critical Systems*, New Orleans, Louisiana, Dec. 1991, pp. 66–76.

[15] Butler, Ricky W.; and Finelli, George B.: The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. *IEEE Transactions on Software Engineering*, vol. 19, no. 1, Jan. 1993, pp. 3–12.

[16] Butler, Ricky W.; and Johnson, Sally C.: Formal Methods For Life-Critical Software. In *Computing in Aerospace 9 Conference*, San Diego, CA, Oct. 1993, pp. 319–329.

[17] Butler, Ricky W.; and Sjogren, Jon A.: *Hardware Proofs Using EHDM and the RSRE Verification Methodology.* NASA Technical Memorandum 100669, Dec. 1988.

[18] Butler, Ricky W., (ed.): *NASA Formal Methods Workshop 1990.* NASA Conference Publication 10052, Nov. 1990.

[19] Carreño, Victor A.; and Angellatta, Rob K.: *A Case Study for the Real-Time Experimental Evaluation of the VIPER Microprocessor.* NASA Technical Memorandum 104098, Sept. 1991.

[20] Computer Resource Management, Inc.: In *Digital Systems Validation Handbook – volume III*, no. DOT/FAA/CT-88/10. FAA.

[21] Courcoubetis, Costas, editor 1993: *Computer Aided Verification, CAV '93*, vol. 697 of *Lecture Notes in Computer Science*, Elounda, Greece, June/July 1993. Springer Verlag.

[22] Di Vito, Ben L.; and Butler, Ricky W.: Provable Transient Recovery for Frame-Based, Fault-Tolerant Computing Systems. In *Real-Time Systems Symposium*, Phoenix, Az, Dec. 1992.

[23] Di Vito, Ben L.; and Butler, Ricky W.: Formal Techniques for Synchronized Fault-Tolerant Systems. In *Dependable Computing for Critical Applications 3*, Dependable Computing and Fault-Tolerant Systems, pp. 279–306. Springer Verlag, Wien New York, 1993. Also presented at 3rd IFIP Working Conference on Dependable Computing for Critical Applications, Mondello, Sicily, Italy, Sept. 14–16, 1992.

[24] Di Vito, Ben L.; Butler, Ricky W.; and Caldwell, James L.: High Level Design Proof of a Reliable Computing Platform. In *Dependable Computing for Critical Applications 2*, Dependable Computing and Fault-Tolerant Systems, pp. 279–306. Springer Verlag, Wien New York, 1992. Also presented at 2nd IFIP Working Conference on Dependable Computing for Critical Applications, Tucson, AZ, Feb. 18–20, 1991, pp. 124–136.

[25] Di Vito, Ben L.; Butler, Ricky W.; and Caldwell, James L., II: *Formal Design and Verification of a Reliable Computing Platform For Real-Time Control (Phase 1 Results)*. NASA Technical Memorandum 102716, Oct. 1990.

[26] Eichenlaub, Carl T.; Harper, C. Douglas; and Hird, Geoffrey: *Using Penelope to Assess the Correctness of NASA Ada Software: A Demonstration of Formal Methods as a Counterpart to Testing.* NASA Contractor Report 4509, May 1993.

[27] Fisher, Gene; Frincke, Deborah; Wolber, Dave; and Cohen, Gerald C.: *Structured Representation for Requirements and Specifications.* NASA Contractor Report 187522, July 1991.

[28] Frincke, Deborah; Wolber, Dave; Fisher, Gene; and Cohen, Gerald: Requirements Specification Language (RSL) and Supporting Tools. Nov. 1992.

[29] Fura, David A.; Windley, Phillip J.; and Cohen, Gerald C.: *Formal Design Specification of a Processor Interface Unit.* NASA Contractor Report 189698, Nov. 1992.

[30] Garmen, John R.: The Bug Heard 'Round The World. *ACM Software Engineering Notes*, vol. 6, no. 5, Oct. 1981, pp. 3–10.

[31] Gibbs, W. Wayt: Software's Chronic Crisis. *Scientific American*, Sept. 1994, pp. 86–95.

[32] Goldberg, Jack; et al.: *Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer.* NASA Contractor Report 172146, 1984.

[33] Guaspari, David: Penelope, an Ada Verification System. In *Proceedings of Tri-Ada '89*, Pittsburgh, PA, Oct. 1989, pp. 216–224.

[34] Guaspari, David: Formally Specifying the Logic of an Automatic Guidance Controller. In *Ada-Europe Conference*, Athens, Greece, May 1991.

[35] Hamilton, Margaret: Zero-defect software: the elusive goal. *IEEE Spectrum*, Mar. 1986.

[36] Harper, Richard E.; Lala, Jay H.; and Deyst, John J.: Fault Tolerant Parallel Processor Architecture Overview. In *Proceedings of the 18th Symposium on Fault Tolerant Computing*, 1988, pp. 252–257.

[37] Hoover, Doug; and Chen, Zewei: *TBell: A Mathematical Tool for Analyzing Decision Tables.* NASA Contractor Report 195027, Nov. 1994.

[38] Houston, Iain; and King, Steve: CICS Project Report: Experiences and Results from the Use of Z in IBM. In Prehn, S.; and Toetenel, W.J., editors 1991:, *VDM '91: Formal Software Development Methods*, Noordwijkerhout, The Netherlands, Oct. 1991, Springer Verlag, pp. 588–596. Volume 1: Conference Contributions.

[39] Hunt, Warren A.: A Formal HDL and its use in the FM9001 Verification. In Hoare, C.A.R.; and Gordon, M.J., editors 1992:, *Mechanized Reasoning in Hardware Design.* Prentice-Hall, 1992.

[40] John Kelly, et. al.: Formal Methods Demonstration Project for Space Applications - Phase I Case Study: Space Shuttle Orbit DAP Jet. Dec. 1993.

[41] Johnson, Sally C.; and Butler, Ricky W.: Design For Validation. In *AIAA/IEEE 10th Digital Avionics Systems Conference*, Los Angeles, California, Oct. 1991, pp. 487–492.

[42] Johnson, Sally C.; and Butler, Ricky W.: Design For Validation. *IEEE Aerospace and Electronics Systems*, Jan. 1992, pp. 38–43.

[43] Johnson, Sally C.; Holloway, C. Michael; and Butler, Ricky W.: *Second NASA Formal Methods Workshop 1992.* NASA Conference Publication 10110, Nov. 1992.

[44] Joyce, Ed: Softwarė Bugs: A Matter of Life and Liability. *Datamation*, May 1987.

[45] Kalvala, Sara; Archer, Myla; and Levitt, Karl: A Methodology for Integrating Hardware Design and Verification. In *ACM International Workshop on Formal Methods in VLSI Design*, Miami, FL, Jan. 1991.

[46] Kalvala, Sara; Levitt, Karl; and Cohen, Gerald C.: *Design and Verification of a DMA Processor.* NASA contractor report, 1992. Unpublished.

[47] Knight, John C.; and Leveson, Nancy G.: An experimental evaluation of the assumptions of independence in multiversion programming. *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, Jan. 1986, pp. 96–109.

[48] Knight, John. C.; and Leveson, Nancy. G.: A Reply To the Criticisms Of The Knight & Leveson Experiment. *ACM SIGSOFT Software Engineering Notes*, Jan. 1990.

[49] Lamport, Leslie: Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 2, Apr. 1984, pp. 254–280.

[50] Lamport, Leslie; and Melliar-Smith, P. M.: Synchronizing Clocks in the Presence of Faults. *Journal Of The ACM*, vol. 32, no. 1, Jan. 1985, pp. 52–78.

[51] Lamport, Leslie; Shostak, Robert; and Pease, Marshall: The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, July 1982, pp. 382–401.

[52] Leveson, Nancy G.: Software Safety: What, Why, and How. *Computing Surveys*, vol. 18, no. 2, June 1986.

[53] Levitt, Karl; and et. al.: *Formal Verification of a Microcoded VIPER Microprocessor using HOL.* NASA Contractor Report 4489, Feb. 1993.

[54] Lincoln, Patrick; and Rushby, John: Formal Verification of an Algorithm for Interactive Consistency under a Hybrid Fault Model. In Courcoubetis [21], pp. 292–304.

[55] Lincoln, Patrick; and Rushby, John: *A Formally Verified Algorithm For Interactice Consistency Under a Hybrid Fault Model.* NASA Contractor Report 4527, July 1993.

[56] Lincoln, Patrick; and Rushby, John: A Formally Verified Algorithm for Interactive Consistency under a Hybrid Fault Model. In *Fault Tolerant Computing Symposium 23*, Toulouse, France, June 1993, IEEE Computer Society, pp. 402–411.

[57] Lincoln, Patrick; and Rushby, John: Formal Verification of an Interactive Consistency Algorithm for the Draper FTP Architecture under a Hybrid Fault Model. In *1994 Computer Assurance (COMPASS) Conference*, June 1994.

[58] Meissner, Charles W., Jr.; Dunham, Janet R.; and (eds.), C. Crim: *Proceedings of the NASA-LaRC Flight-Critical Digital Systems Technology Workshop.* NASA Conference Publication 10028, Apr. 1989.

[59] Miller, Steve; and Srivas, Mandayam: Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods. In *WIFT'95 Workshop on Industrial-strength Formal Specification Techniques*, Boca Raton, Florida USA, Apr. 1995. To appear.

[60] Miner, Paul S.: *An Extension to Schneider's General Paradigm for Fault-Tolerant Clock Synchronization.* NASA Technical Memorandum 107634, Langley Research Center, Hampton, VA, 1992.

[61] Miner, Paul S.: *A Verified Design of a Fault-Tolerant Clock Synchronization Circuit: Preliminary Investigations.* NASA Technical Memorandum 107568, Mar. 1992.

[62] Miner, Paul S.: *Verification of Fault-Tolerant Clock Synchronization Systems.* NASA Technical Paper 3349, Nov. 1993.

[63] Miner, Paul S.; Padilla, Peter A.; and Torres, Wilfredo: A Provably Correct Design of a Fault-Tolerant Clock Synchronization Circuit. In *11th Digital Avionics Systems Conference*, Seattle, WA, Oct. 1992, pp. 341–346.

[64] Miner, Paul S.; Pullela, Shyamsundar; and Johnson, Steven D.: Interaction of Formal Design Systems in the Development of a Fault-Tolerant Clock Synchronization Circuit. In *13th Symposium on Reliable Distributed Systems.* IEEE Computer Society Press, 1994, pp. 128–137. Proceedings of SRDS 94 held at Dana Point, California, October 1994.

[65] Moore, J Strother: *A Formal Model of Asynchronous Communication and Its Use in Mechanically Verifying a Biphase Mark Protocol.* NASA Contractor Report 4433, June 1992.

[66] Moore, J Strother: *Mechanically Verified Hardware Implementing an 8-bit Parallel IO Byzantine Agreement Processor.* NASA Contractor Report 189588, Apr. 1992.

[67] Neumann, Peter G.: Some Computer-Related Disasters and Other Egregious Horrors. *ACM Software Engineering Notes*, vol. 10, no. 1, Jan. 1985, pp. 6–12.

[68] Owre, S.; Shankar, N.; and Rushby, J. M.: *The PVS Specification Language (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[69] Owre, S.; Shankar, N.; and Rushby, J. M.: *User Guide for the PVS Specification and Verification System (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[70] Owre, Sam; Rushby, John; ; Shankar, Natarajan; and von Henke, Friedrich: Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 1995. To appear.

[71] Pan, Jing; and Levitt, Karl: Towards a Formal Specification of the IEEE Floating-Point Standard with Application to the Verification of Floating-Point Coprocessors. In *24th Asilomar Conference on Signals, Systems & Computers*, Monterrey, CA., Nov. 1990.

[72] Pan, Jing; Levitt, Karl; and Cohen, Gerald C.: *Toward a Formal Verification of a Floating-Point Coprocessor and its Composition with a Central Processing Unit.* NASA Contractor Report 187547, Aug. 1991.

[73] Pan, Jing; Levitt, Karl; and Schubert, E. Thomas: Toward a Formal Verification of a Floating-Point Coprocessor and its Composition with a Central Processing Unit. In *ACM International Workshop on Formal Methods in VLSI Design*, Miami, FL, Jan. 1991.

[74] Rogers, Michael; and Gonzalez, David L.: Can We Trust Our Software? *Newsweek*, Jan. 1990.

[75] Rushby, John: *Formal Specification and Verification of a Fault-Masking and Transient-Recovery Model for Digital Flight-Control Systems.* NASA Contractor Report 4384, July 1991.

[76] Rushby, John: *Formal verification of an Oral Messages algorithm for interactive consistency.* NASA Contractor Report 189704, Oct. 1992.

[77] Rushby, John: *Formal Methods and Digital Systems Validation for Airborne Systems.* NASA Contractor Report 4551, 1993.

[78] Rushby, John: A Formally Verified Algorithm Clock Sychronization Under a Hybrid Fault Model. In *ACM Principles of Distributed Computing '94*, Aug. 1994.

[79] Rushby, John; and von Henke, Friedrich: *Formal Verification of a Fault-Tolerant Clock Synchronization Algorithm.* NASA Contractor Report 4239, June 1989.

[80] Rushby, John; and von Henke, Friedrich: Formal Verification of Algorithms for Critical Systems. *IEEE Transactions on Software Engineering*, vol. 19, no. 1, Jan. 1993, pp. 13–23.

[81] Schneider, Fred B.: *Understanding Protocols for Byzantine Clock Synchronization.* Cornell University, Ithaca, NY, Technical Report 87-859, Aug. 1987.

[82] Schrage, Michael: 'When the Chips Are Down' Will Likely Be Heard More Often in Computing. *The Washington Post*, pp. B3. December 16, 1994.

[83] Schubert, Thomas; and Levitt, Karl: Verification of Memory Management Units. In *Second IFIP Conference on Dependable Computing For Critical Applications*, Tucson, Arizona, Feb. 1991, pp. 115–123.

[84] Schubert, Thomas; Levitt, Karl; and Cohen, Gerald C.: *Towards Composition of Verified Hardware Devices.* NASA Contractor Report 187504, Nov. 1991.

[85] Schubert, Thomas; Levitt, Karl; and Cohen, Gerald C.: *Formal Mechanization of Device Interactions With a Process Algebra.* NASA Contractor Report 189644, Nov. 1992.

[86] Schubert, Thomas; Levitt, Karl; and Cohen, Gerald C.: *Formal Verification of a Set of Memory Management Units.* NASA Contractor Report 189566, 1992.

[87] Shankar, N.; Owre, S.; and Rushby, J. M.: *The PVS Proof Checker: A Reference Manual (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[88] Shankar, Natarajan: *Mechanical Verification of a Schematic Byzantine Clock Synchronization Algorithm.* NASA Contractor Report 4386, July 1991.

[89] Shankar, Natarajan: Mechanical Verification of a Generalized Protocol for Byzantine Fault-Tolerant Clock Synchronization. In *Second International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, vol. 571 of *Lecture Notes in Computer Science*, pp. 217–236. Springer Verlag, Nijmegen, The Netherlands, Jan. 1992.

[90] Shankar, Natarajan: Verification of Real-Time Systems Using PVS. In Courcoubetis [21], pp. 280–291.

[91] Shankar, Natarajan; Owre, Sam; and Rushby, John: *PVS Tutorial.* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

[92] Srivas, Mandayam; and Bickford, Mark: *Verification of the FtCayuga Fault-Tolerant Microprocessor System (Volume 1: A Case Study in Theorem Prover-Based Verification).* NASA Contractor Report 4381, July 1991.

[93] Srivas, Mandayam; and Bickford, Mark: *Moving Formal Methods Into Practice: Verifying the FTPP Scoreboard: Phase 1 Results.* NASA Contractor Report 189607, May 1992.

[94] Thambidurai, Philip; and Park, You-Keun: Interactive Consistency With Multiple Failure Modes. In *7th Symposium on Reliable Distributed Systems*, Columbus, OH, Oct. 1988, pp. 93–100.

[95] Wiener, Lauren Ruth: *Digital Woes.* Addison-Wesley Publishing Company, 1993. ISBN 0-201-62609-8.

[96] Windley, Phillip J.: Abstract Hardware. In *ACM International Workshop on Formal Methods in VLSI Design*, Miami, FL, Jan. 1991.

[97] Windley, Phillip J.: The Formal Verification of Generic Interpreters. In *28th Design Automation Conference*, San Franciso, CA, June 1991.

[98] Windley, Phillip J.; Levitt, Karl; and Cohen, Gerald C.: *Formal Proof of the AVM-1 Microprocessor Using the Concept of Generic Interpreters.* NASA Contractor Report 187491, Mar. 1991.

[99] Windley, Phillip J.; Levitt, Karl; and Cohen, Gerald C.: *The Formal Verification of Generic Interpreters.* NASA Contractor Report 4403, Oct. 1991.

[100] Young, William D.: *Verifying the Interactive Convergence Clock Synchronization Algorithm Using the Boyer-Moore Theorem Prover.* NASA Contractor Report 189649, Apr. 1992.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*

**1. AGENCY USE ONLY** ***(Leave blank)***

**2. REPORT DATE**
June 1995

**3. REPORT TYPE AND DATES COVERED**
Conference Publication

**4. TITLE AND SUBTITLE**
Third NASA Langley Formal Methods Workshop

**5. FUNDING NUMBERS**
WU 505-64-50-03
WU 505-64-10-13

**6. AUTHOR(S)**
C. Michael Holloway, Compiler

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
NASA Langley Research Center
Hampton, VA 23681-0001

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
National Aeronautics and Space Administration
Washington, DC 20546-0001

**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
NASA CP-10176

**11. SUPPLEMENTARY NOTES**
This workshop was chaired by Ricky W. Butler and C. Michael Holloway of NASA Langley Research Center. Administrative chair was Lisa F. Peckham, also of NASA Langley Research Center.

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**
Unclassified-Unlimited

Subject Category 59

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** ***(Maximum 200 words)***

This publication constitutes the proceedings of NASA Langley Research Center's third workshop on the application of formal methods to the design and verification of life-critical systems. This workshop brought together formal methods researchers, industry engineers, and academicians to discuss the potential of NASA-sponsored formal methods and to investigate new opportunities for applying these methods to industry problems. Contained herein are copies of the material presented at the workshop, summaries of many of the presentations, a complete list of attendees, and a detailed summary of the Langley formal methods program. Much of this material is available electronically through the World-Wide Web via the following URL: http://atb-www.larc.nasa.gov/WS95/proceedings.html.

**14. SUBJECT TERMS**
formal methods; specification; design; verification; mathematical modeling; technology transfer; formal logic

**15. NUMBER OF PAGES**
269

**16. PRICE CODE**
A12

**17. SECURITY CLASSIFICATION OF REPORT**
Unclassified

**18. SECURITY CLASSIFICATION OF THIS PAGE**
Unclassified

**19. SECURITY CLASSIFICATION OF ABSTRACT**

**20. LIMITATION OF ABSTRACT**