NASA Conference Publication 10052

NASA Formal Methods Workshop 1990

Compiled by
Ricky W. Butler
NASA 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 August 20–23, 1990

November 1990

NASA
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665-5225
Contents

Introduction ................................................................................. 1

Workshop Agenda ........................................................................... 2

List of Attendees ........................................................................... 5

Digital Avionics a Cornerstone of Avionics
by Chuck Meissner for Cary Spitzer (NASA) ........................................ 6

Life Critical Digital Flight Control Systems
by James McWha (Boeing Military) ................................................. 30

Advanced Embedded Processing: Present and Future
by Jerry Cohen (Boeing Military) .................................................... 52

MAFT: The Multicomputer Architecture for Fault Tolerance
by Roger Kieckhafer (U. Nebraska, Lincoln) .................................... 94

Design For Validation
by Rick Butler (NASA) .................................................................. 146

What FM can offer to DFCS design
by John Rushby (SRI International) .............................................. 165

What FM can offer to DFCS design
by Don Good (Computational Logic, Inc) ...................................... 195

High Level Design Proof of a Reliable Computing Platform
Ben DiVito (Vigyan) ....................................................................... 232

A Verified Model of Fault Tolerance
John Rushby (SRI International) ................................................... 260

The Design and Verification of a Fault-tolerant Circuit
Bill Bevier and Bill Young (Computational Logic, Inc) ....................... 280

Verifying an Interactive Consistency Circuit
Hardware Verification at Computational Logic Inc.
by Warren Hunt (Computational Logic, Inc) ........................................ 325

Generic Interpreters and Microprocessor Verification
by Phil Windley (Univ. California, Davis) ........................................... 347

VIPER Project
by Clive Pygott and John Kershaw (Royal Signals and Radar Establishment) ...... 380

Mechanical Proofs of Fault-Tolerant Clock Synchronization
by N. Shankar and John Rushby (SRI International) .................................. 405

An HOL Theory For Voting
by Paul Miner and Jim Caldwell (NASA) .............................................. 442

Formally Specifying the Logic Of Automatic Guidance Control (Ada)
by David Guaspari (Odyssey Research Associates) .................................. 457

Verification of Floating-point Software
by Doug Hoover (Odyssey Research Associates) .................................... 477

C Formal Verification with Unix Communication and Concurrency
by Doug Hoover (Odyssey Research Associates) .................................... 495
Introduction

This publication contains copies of the material presented at the NASA Formal Methods Workshop held at Langley Research Center on August 20-23, 1990. The purpose of the workshop was to bring together the researchers involved in the NASA formal methods research effort for detailed technical interchange and to provide a chance for interaction with representatives from the U.S. government and the aerospace industry. The goals of the workshop were:

- Introduce the formal methods research teams to a broader view of the aerospace problem domain by industry presentations.

- Detailed technical exchange between formal methods research teams to define and characterize the verification problem for ultra-reliable life-critical flight control systems.

- Identification of aerospace problems which can benefit from formal methods and can serve as the basis of future research efforts.

The NASA effort in formal methods includes researchers at NASA LaRC, Computational Logic Inc., Odyssey Research Associates, SRI International, Boeing Military, Vigyan and the University of California at Davis and Irvine. Also NASA Langley is involved in a joint research effort with the UK Royal Signals and Radar Establishment as formalized in a Memorandum of Understanding between the two organizations.

Attendees at the workshop included NASA personnel, researchers from the four supporting contract organizations, RSRE personnel, invited speakers, and representatives from other government research organizations with interests in formal methods. Attendance was by invitation only.
NASA Formal Methods Workshop Agenda
(Aug 20-23, 1990)

Day 1

8:00 - 8:20 am  Late Registration
8:20 - 8:30 am  Greeting by Chief of ISD
8:30 - 8:45 am  Workshop Objectives
8:45 - 9:30 am  Digital Avionics: A Cornerstone of Aviation

9:30 - 10:15 am  Life Critical Digital Flight Control Systems (DFCS)
10:15 - 10:30 am  BREAK

10:30 - 11:30 am  Advanced Embedded Processing: Present and Future

11:30 - 12:30 am  LUNCH

12:30 - 1:30 pm  MAFT: The Multicomputer Architecture for Fault Tolerance
1:30 - 2:00 pm  Design For Validation
2:00 - 2:30 pm  BREAK

2:30 - 3:00 pm  What FM can offer to DFCS design
3:00 - 3:30 pm  What FM can offer to DFCS design
3:30 - 4:00 pm  What FM can offer to DFCS design

7:00 pm  Conference Dinner
<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
<th>Speaker(s)</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 - 9:30 am</td>
<td>OS verification</td>
<td>DiVito</td>
<td>High Level Design Proof of a Reliable Computing Platform</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rushby</td>
<td>A Verified Model of Fault Tolerance</td>
</tr>
<tr>
<td>10:15 - 10:30 am</td>
<td>BREAK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:30 - 11:30 pm</td>
<td>Byzantine Generals</td>
<td>Bevier &amp; Young</td>
<td>The Design and Verification of a Fault-tolerant Circuit</td>
</tr>
<tr>
<td>11:30 - 12:30 am</td>
<td>LUNCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12:30 - 1:30 am</td>
<td>Verifying an Interactive Consistency Circuit</td>
<td>Srivas</td>
<td></td>
</tr>
<tr>
<td>1:30 - 2:00 pm</td>
<td>Hardware Verification at Computational Logic Inc.</td>
<td>Hunt</td>
<td></td>
</tr>
<tr>
<td>2:00 - 2:30 pm</td>
<td>Generic Interpreters and Microprocessor Verification</td>
<td>Windley</td>
<td></td>
</tr>
<tr>
<td>2:30 - 3:00 pm</td>
<td>BREAK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:00 - 4:00 pm</td>
<td>VIPER 2 &amp; NODEN</td>
<td>Pygott/Kershaw</td>
<td></td>
</tr>
<tr>
<td>4:00 - 4:30 pm</td>
<td>Discussion</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Day 3

8:30 - 10:00 am Shankar/Rushby
10:00 - 10:30 am Discussion
10:30 - 11:30 pm Levitt
11:30 - 12:00 pm Caldwell/Carreno/Miner
12:00 - 1:00 pm LUNCH
1:00 - 1:45 pm Guaspari
1:45 - 2:30 pm Hoover
2:30 - 3:00 pm Hoover
3:00 - 3:30 pm BREAK
3:30 - 5:00 pm Planning

--- Clock Synchronization ---
Mechanical Proofs of Fault-Tolerant Clock Synchronization

--- Commercial Chips ---
Floating-pt. Coprocessor (Intel 8087)
DMA controller (Intel 8237A), etc.
(Not in Proceedings)
An HOL Theory For Voting

--- Code Verification ---
Formally Specifying the Logic Of
Automatic Guidance Control (Ada)
Verification of Floating-point Software
C Formal Verification with Unix Communication and Concurrency

Day 4

8:30 - 12:00 pm Discussion
**NASA FM Workshop Attendees**

<table>
<thead>
<tr>
<th>Name</th>
<th>Email</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bill Bevier</td>
<td><a href="mailto:bevier@cli.com">bevier@cli.com</a></td>
<td>(512)322-9951</td>
</tr>
<tr>
<td>Deepak Kapur</td>
<td><a href="mailto:kapur@albanycs.albany.edu">kapur@albanycs.albany.edu</a></td>
<td>(518)442-4281</td>
</tr>
<tr>
<td>Dale Johnson</td>
<td></td>
<td>(617)271-8894</td>
</tr>
<tr>
<td>Andy Moore</td>
<td></td>
<td>(202)767-6698</td>
</tr>
<tr>
<td>Karl Levitt</td>
<td></td>
<td>(916)752-0832</td>
</tr>
<tr>
<td>Sally Johnson</td>
<td></td>
<td>(804)864-6204</td>
</tr>
<tr>
<td>Richard Platek</td>
<td></td>
<td>(607)277-2020</td>
</tr>
<tr>
<td>Bill Young</td>
<td></td>
<td>(512)322-9951</td>
</tr>
<tr>
<td>Don Good</td>
<td></td>
<td>(512)322-9951</td>
</tr>
<tr>
<td>Warren Hunt</td>
<td></td>
<td>(804)864-6214</td>
</tr>
<tr>
<td>Jim Caldwell</td>
<td></td>
<td>(607)277-2020</td>
</tr>
<tr>
<td>Mark Bickford</td>
<td></td>
<td>(613)238-7900</td>
</tr>
<tr>
<td>Mark Saaltink</td>
<td></td>
<td>(301)688-4229</td>
</tr>
<tr>
<td>Bill Legato</td>
<td></td>
<td>(313)763-0391</td>
</tr>
<tr>
<td>Kang Shin</td>
<td><a href="mailto:kgshin@eecs.umich.edu">kgshin@eecs.umich.edu</a></td>
<td>(804)864-6218</td>
</tr>
<tr>
<td>Chuck Meissner</td>
<td></td>
<td>(609)484-5577</td>
</tr>
<tr>
<td>Pete Saraceni</td>
<td></td>
<td>(402)472-2402</td>
</tr>
<tr>
<td>Roger Kieckhafer</td>
<td></td>
<td>(607)277-2020</td>
</tr>
<tr>
<td>Doug Hoover</td>
<td></td>
<td>(916)752-6452</td>
</tr>
<tr>
<td>David Guaspari</td>
<td><a href="mailto:rogerk@fergvax.unl.edu">rogerk@fergvax.unl.edu</a></td>
<td>(402)472-2402</td>
</tr>
<tr>
<td>Tom Schubert</td>
<td></td>
<td>(607)277-2020</td>
</tr>
<tr>
<td>Mark Ardis</td>
<td></td>
<td>(804)864-6218</td>
</tr>
<tr>
<td>Clive Pygott</td>
<td></td>
<td>(609)484-5577</td>
</tr>
<tr>
<td>John Kershaw</td>
<td></td>
<td>(402)472-2402</td>
</tr>
<tr>
<td>David Musser</td>
<td>hoove%<a href="mailto:oravax.uucp@cu-arpa.cs.cornell.edu">oravax.uucp@cu-arpa.cs.cornell.edu</a></td>
<td>(607)277-2020</td>
</tr>
<tr>
<td>Phil Windley</td>
<td></td>
<td>(916)752-6452</td>
</tr>
<tr>
<td>John Knight</td>
<td></td>
<td>(402)472-2402</td>
</tr>
<tr>
<td>Rick Kuhn</td>
<td><a href="mailto:schubert@iris.ucdavis.edu">schubert@iris.ucdavis.edu</a></td>
<td>(607)277-2020</td>
</tr>
<tr>
<td>Dave Eckhardt</td>
<td></td>
<td>(301)975-3337</td>
</tr>
<tr>
<td>John McHugh</td>
<td><a href="mailto:mchugh@cli.com">mchugh@cli.com</a></td>
<td>(804)864-1698</td>
</tr>
<tr>
<td>Milt Holt</td>
<td></td>
<td>(919)493-4932</td>
</tr>
<tr>
<td>Ben DiVito</td>
<td><a href="mailto:bld@air16.larc.nasa.gov">bld@air16.larc.nasa.gov</a></td>
<td>(804)864-1596</td>
</tr>
<tr>
<td>Paul Miner</td>
<td><a href="mailto:psm@air16.larc.nasa.gov">psm@air16.larc.nasa.gov</a></td>
<td>(804)864-4883</td>
</tr>
<tr>
<td>Rick Butler</td>
<td><a href="mailto:rwb@air16.larc.nasa.gov">rwb@air16.larc.nasa.gov</a></td>
<td>(804)864-6201</td>
</tr>
<tr>
<td>M.K. Srivas</td>
<td>srivas%<a href="mailto:oravax.uucp@cu-arpa.cs.cornell.edu">oravax.uucp@cu-arpa.cs.cornell.edu</a></td>
<td>(804)864-6198</td>
</tr>
<tr>
<td>John Rushby</td>
<td><a href="mailto:rushby@csl.sri.com">rushby@csl.sri.com</a></td>
<td>(607)277-2020</td>
</tr>
<tr>
<td>N. Shankar</td>
<td><a href="mailto:shankar@csl.sri.com">shankar@csl.sri.com</a></td>
<td>(415)859-5456</td>
</tr>
<tr>
<td>Gerry Cohen</td>
<td></td>
<td>(206)865-3739</td>
</tr>
</tbody>
</table>

NOT ON INTERNET
DIGITAL AVIONICS

A CORNERSTONE OF AVIATION

by

Cary R. Spitzer

NASA Langley Research Center

Presented to the NASA Formal Methods Workshop

by

Charles W. Meissner, Jr.
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

INTRODUCTION: Avionics Roles

- Communication
  - HF and VHF
  - Satellite
  - Data Links

- Navigation
  - Ground-based systems
  - Inertial and satellite-based systems
  - Goal: Autonomous operation!!!
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

INTRODUCTION

CURRENT EXAMPLES

CURRENT ISSUES

FUTURE TRENDS

INTERNATIONAL SCENE

SUMMARY
FIGHTER INSTALLED AVIONICS WEIGHT

Installed Weight, Lbs.


Years
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

INTRODUCTION: Avionics Roles

- Fly-by-wire flight controls
  - Historically used for stability & control augmentation
    - Not flight critical

- Emerging as a flight critical system
  - Driven by performance and economic demands
    - F-16, A-320, B-777
TOTAL ON BOARD COMPUTER CAPACITY (OFP)
F-16 AVIONICS SYSTEM ARCHITECTURE

- Engine
- Global Positioning System
- Data Transfer Unit
- Combined Altitude Radar Altimeter
- Central Air Data Computer
- AMUX (1553)
- Up-Front Controls
- Crash-Survivable Flight Data Recorder
- Inertial Navigation System
- BMUX (1553)
- Hard Points
- Advanced Self-Protection Jammer
- Advanced Interference Blanker Unit
- Advanced IFF System
- Fire Control Radar (APG-68)
- Digital Flight Control Computer
- CMUX (1553)
- Store Stations
- Chaff/Flare Dispenser (ALE-47)
- Advanced Radar Warning Receiver
- DMUX (1553)
- Programmable Display Generator
- Head-Up Display Electronic Unit
- Multi-Function Displays
- Head-Up Display
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

CURRENT EXAMPLES: A-320

Pitch Control
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

CURRENT EXAMPLES: A-320

Electronic Flight Control System Architecture
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

CURRENT EXAMPLES: A-320

**Yaw Control**

<table>
<thead>
<tr>
<th>M</th>
<th>Motor actuator</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAC</td>
<td>Flight augmentation computer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hydraulic</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
</tr>
<tr>
<td>G</td>
</tr>
<tr>
<td>Y</td>
</tr>
</tbody>
</table>
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

CURRENT EXAMPLES: A-320

Hydraulic actuators

RIGHT
Ailerons

LEFT

Autopilot commands

ELAC

(1) (2)

LAF

SEC

(1) (2) (3)

Spoilers 2, 3, 4 & 5

Sidestick commands

LAF Load alleviation function
ELAC Elevator and aileron computer
SEC Spoiler and elevator computer

Hydraulic
B Blue system
G Green system
Y Yellow system

Roll Control
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

CURRENT ISSUES: Hardware

- Modeling of complex systems
  - Proof of fault tolerance, high reliability

- Electromagnetic interference
  - Growing concern due to composite aircraft, increased emission of RF, and smaller electronic element sizes
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

FUTURE TRENDS: PAVE PILLAR
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

CURRENT ISSUES: Software

- Developing competency in Ada
  - Mandated for DoD, Space Station Freedom, civil transports
- Computer-Aided Software Engineering (CASE) Tools
  - Capabilities for real-time software analysis & design
  - Tool validation
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

FUTURE TRENDS: INTEGRATED MODULAR AVIONICS
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

FUTURE TRENDS: PAVE PILLAR
DIGITAL AVIONICS: A CORNERSTONE OF AVIATION

FUTURE TRENDS: Supporting Technologies

- Flat panel, full color, liquid crystal displays
- Replacing CRTs
- Advanced formats; not electronic steam gauges
- Higher speed data buses
- Artificial intelligence pioneer programs
- Faultfinder
- Diverter
- Pilot's Associate
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

FUTURE TRENDS: INTEGRATED MODULAR AVIONICS

HARDWARE

- Datac Terminal
- Low Voltage Power Supplies
- Central Processor
- 'Standard' I/O
- Custom Sensors

SOFTWARE

- Executive
- BITE (Hardware)
- BITE (System)
- Application

COMMON COMPONENTS

APPLICATION SPECIFIC COMPONENTS

COMPONENTS OF TYPICAL LRU
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

INTERNATIONAL SCENE: Japan

- An emerging competitor in the world market

- Historically has been component oriented: displays, microprocessors, etc.

- Lack system design and analysis, & software capabilities
  - FS-X program will help to build a foundation for military & civil avionics

- MITI has established a committee to define an avionics technology development plan
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

INTERNATIONAL SCENE: Europe

- Leading firms are GEC Avionics, Smiths Industries, Sextant, & Aerospatiale
- Extremely capable; serious competition for U.S. firms
  - Build most of the Airbus avionics
- GEC Avionics will build the B-777 flight control system
  - Build flight controls for Jaguar and YC-14
- European Community 92 will strengthen competitive threat
DIGITAL AVIONICS - A CORNERSTONE OF AVIATION

SUMMARY

- Continually expanding role for avionics
- Flight critical avionics are here
- Strong emphasis on Ada
- Module-based architectures emerging
- Artificial intelligence applications being developed
- Significant competitive threat to U.S. firms from Europe & Japan
LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

JIM McWHA

CHIEF ENGINEER - FLIGHT CONTROLS
BOEING COMMERCIAL AIRPLANE GROUP

AUGUST 20, 1990
LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

INDUSTRY STATUS

DIGITAL AUTOPILOT SYSTEMS WERE FIRST CERTIFICATED FOR USE ON COMMERCIAL AIRPLANES IN THE LATE 1970'S

THE A-320 AIRPLANE WAS THE FIRST COMMERCIAL AIR TRANSPORT AIRPLANE TO BE CERTIFICATED WITH A FLY BY WIRE PRIMARY FLIGHT CONTROL SYSTEM

BOEING WILL HAVE ALL FLY BY WIRE FLIGHT CONTROLS ON THE 767-X (777) AIRPLANE
LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

- DEFINITION
- SAFETY
- INDUSTRY STATUS
- PROGRAM PHASES
LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

DEFINITION

A CONTROL SYSTEM IMPLEMENTED IN DIGITAL COMPUTER TECHNOLOGY WHICH HAS A FUNCTION WHICH IF NOT PERFORMED AS INTENDED IS LIFE THREATENING

EXAMPLES:  AN AUTOPilot USED FOR AUTOMATIC LANDING IN LOW VISIBILITY CONDITIONS

AN AIRPLANE CONTROL SYSTEM IMPLEMENTED WITHOUT CONTROL CABLES:

FLY BY WIRE

FLY BY LIGHT
767-X PRIMARY FLIGHT CONTROL SYSTEM OVERVIEW
- DISSIMILAR CHANNELS - LEFT, CENTER, RIGHT
- SIMILAR LANES IN EACH CHANNEL - ONE IN COMMAND, TWO FUNCTIONING AS MONITORS
- TWO IN E/E BAY, ONE AFT OF FWD CARGO DOOR

ARINC 629 BUSES

PRIMARY FLIGHT COMPUTER ARCHITECTURE
767-X ELECTRICAL POWER SYSTEM
FLIGHT CRITICAL DC

- PMGs
- PMGs
- PMGs

- L Engine
- APU
- R Engine

- L 28 VDC Bus
- Hot Battery Bus
- R 28 VDC Bus

- FBW PMG Converter (typical)
- FBW Power Supply Assy (typical)
- 5 min. battery (typical)

C FBW BUS

L FBW BUS
R FBW BUS
LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

SAFETY

FEDERAL AVIATION ADMINISTRATION (FAA) REGULATIONS DEFINE THE BASIC SAFETY CRITERIA:

FAR 25.1309 NO SINGLE FAILURE OR COMBINATION OF FAILURES WHICH ARE NOT SHOWN TO BE EXTREMELY IMPROBABLE SHALL PREVENT CONTINUED SAFE FLIGHT AND LANDING OF THE AIRPLANE

EXTREMELY IMPROBABLE - PROBABILITY OF $1 \times 10^{-9}$ OR LESS PER FLIGHT HOUR OR EVENT
LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

SAFETY

FEDERAL AVIATION ADMINISTRATION (FAA) REGULATIONS DEFINE THE BASIC SAFETY CRITERIA:

FAR 25.1309  NO SINGLE FAILURE OR COMBINATION OF FAILURES WHICH ARE NOT SHOWN TO BE EXTREMELY IMPROBABLE SHALL PREVENT CONTINUED SAFE FLIGHT AND LANDING OF THE AIRPLANE

EXTREMELY IMPROBABLE - PROBABILITY OF $1 \times 10^{-9}$ OR LESS PER FLIGHT HOUR OR EVENT
FIGURE 1.2-1 PRIMARY FLIGHT CONTROLS DEVELOPMENT OVERVIEW
# LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

## PROGRAM PHASES - REQUIREMENTS DEFINITION

**TOP DOWN STRUCTURED PROCESS:**

<table>
<thead>
<tr>
<th>AIRPLANE LEVEL REQUIREMENTS</th>
<th>TOP LEVEL DESIGN REQUIREMENTS AND OBJECTIVES</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM REQUIREMENTS</td>
<td>CERTIFICATION REQUIREMENTS</td>
</tr>
<tr>
<td></td>
<td>FUNCTIONAL REQUIREMENTS</td>
</tr>
<tr>
<td></td>
<td>INTEGRITY REQUIREMENTS</td>
</tr>
<tr>
<td></td>
<td>ARCHITECTURAL CONSIDERATIONS</td>
</tr>
<tr>
<td>SOFTWARE REQUIREMENTS</td>
<td>EXPANSION OF SYSTEM REQUIREMENTS TO A LEVEL WHICH CAN BE IMPLEMENTED IN A TARGET DIGITAL COMPUTER OR COMPUTERS</td>
</tr>
</tbody>
</table>
LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

PROGRAM PHASES - REQUIREMENTS DEFINITION (CONT)

REQUIREMENTS WHICH EVOLVE OUT OF THE USE OF SPECIFIC HARDWARESOFTWARE TO IMPLEMENT THE SOFTWARE REQUIREMENTS

DETAILED DESIGN REQUIREMENTS
# LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

## PROGRAM PHASES - DESIGN AND DEVELOPMENT

<table>
<thead>
<tr>
<th>HARDWARE SELECTION</th>
<th>I/O REQUIREMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PROCESSING SPEED</td>
</tr>
<tr>
<td></td>
<td>MEMORY SIZE</td>
</tr>
<tr>
<td></td>
<td>ETC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PROGRAMMING LANGUAGE</th>
<th>INDUSTRY/COMPANY STANDARD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SUPPORT SOFTWARE AVAILABILITY AND MATURITY</td>
</tr>
<tr>
<td></td>
<td>LONG TERM MAINTENANCE</td>
</tr>
<tr>
<td></td>
<td>ETC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CODE GENERATION</th>
<th>TYPICALLY AN INCREMENTAL BUILD PROCESS</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>TESTING</th>
<th>HARDWARE - QUALIFICATION TESTING - RTCA DO-160</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INCREMENTAL SOFTWARE LOADS - VENDOR AND AIRFRAME</td>
</tr>
<tr>
<td></td>
<td>SYSTEMS INTEGRATION / IRON BIRD</td>
</tr>
<tr>
<td></td>
<td>AIRPLANE - GROUND AND FLIGHT</td>
</tr>
</tbody>
</table>
LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

PROGRAM PHASES - VERIFICATION

GUIDELINE DOCUMENT RTCA DOCUMENT DO-178A

VERIFICATION PROCESSES ARE A FUNCTION OF SYSTEM CRITICALITY

CRITICAL SYSTEM A FORMAL PROCESS OF ASSURING THAT ALL SOFTWARE REQUIREMENTS HAVE BEEN IMPLEMENTED COMPLETELY AND EXCLUSIVELY
LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

PROGRAM PHASES - VALIDATION

A PROCESS OF ASSURING THAT ALL SYSTEM REQUIREMENTS HAVE BEEN IMPLEMENTED CORRECTLY

0 ANALYSES

<table>
<thead>
<tr>
<th>SAFETY ANALYSIS</th>
<th>HAZARD ASSESSMENT AND FAILURE ANALYSIS TO ASSURE THAT REQUIREMENTS OF FAR 25.1309 ARE SATISFIED</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERFORMANCE ANALYSIS</td>
<td>ASSURANCE THAT SYSTEM PERFORMS INTENDED FUNCTION WITHIN ACCEPTABLE LIMITS UNDER ALL ALLOWABLE ENVIRONMENTAL AND TOLERANCE CONDITIONS</td>
</tr>
</tbody>
</table>
LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

PROGRAM PHASES - VALIDATION (CONT)

- LABORATORY TESTING  
  Test to isolate errors and problems before flight test. Test under normal and failure conditions.

- SYSTEMS INTEGRATION TESTING  
  Test with as many interfacing systems as possible to ensure compatibility.

- AIRPLANE GROUND TESTING  
  Check of systems installed in an airplane including EMI/HIRF tests.

- AIRPLANE FLIGHT TESTING  
  Comprehensive test of performance in flight under a variety of conditions used to cross check simulation results - autoland system could require 200-300 landings over an 8 month period.
LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

PROGRAM PHASES - CERTIFICATION

THE PROCESS OF DEMONSTRATING TO THE REGULATORY AUTHORITIES THAT ALL SAFETY AND PERFORMANCE REQUIREMENTS ARE SATISFIED

STARTS WITH A CERTIFICATION PLAN WHICH:

IDENTIFIES REGULATIONS AND ACCEPTABLE MEANS OF COMPLIANCE METHODS

DESCRIPTS PROPOSED METHODS OF ESTABLISHING COMPLIANCE

DESCRIPTS THE METHODS AND PROCESSES TO BE USED TO ASSURE AN ORDERLY AND CONTROLLED DESIGN AND DEVELOPMENT PROCESS

FOLLOW ON SPECIALIST MEETINGS

PERFORMANCE AND INTEGRITY DEMONSTRATIONS
LIFE-CRITICAL DIGITAL FLIGHT CONTROL SYSTEMS

PROGRAM PHASES - CERTIFICATION (CONT)

CERTIFICATION SUMMARY

CONFIRMS COMPLETE IMPLEMENTATION OF THE PROCESSES IDENTIFIED IN THE CERTIFICATION PLAN

PROVIDES A MEANS FOR ESTABLISHING VERIFICATION AND VALIDATION COVERAGE
767-X PFCS Schedule


Type Cert. Appl. FAA Interim Type Board FAA Interim Type Board Rollout Type Cert.


Certification Documentation

767-X

757 PFCS Testing

PRE-PRODUCTION DEVELOPMENT Iron Bird Test 757 Flight Test

Production Design Validation

767-X PFCS

Build 767-X Units Iron Bird Test 767-X FLIGHT TEST
Advanced Embedded Processing
Present and Future

The Boeing Company
G.C. Cohen
Integrated Airframe/Propulsion Control System Architecture (IAPSA II)

Began: July 26, 1985
Ended: April 1, 1990
Goals of the Program

1. Design and validation methodology for systems architecture
2. Critical validation issues simulated in Airlab
3. System design
4. System specification
5. Small-scale system testing
Methodology
Why a Methodology

Boeing Military Airplanes

Architecture topology of past

Commercial aircraft
Problem

- Interrelationships difficult to specify in terms of meaningful requirements
  - Normal mode
  - Failure mode
- Unless contractor/vendor team takes a systems approach, system will be overdesigned and still may not meet the requirements
Methodology Elements

- Requirements
- Specifications
- Design
  - Automation
  - Guidelines
  - Building blocks
- Reliability
- Performance
- Design for validation
- Design for cost
- Proof of correctness
- Testing
- Traceability
- Availability
- Survivability
- Maintainability
IAPSA II Prevalidation Methodology

1. Mission requirements
2. System functions requirements
3. Performance criteria
4. Criticality criteria
5. Cost
6. Concept refinement
7. Maintenance

Validation of functional requirements

Redundancy management algorithms

Control law development

System candidate architecture

Modify conceptual definition

Candidate analysis using reliability tools

Candidate analysis using performance tools

Architecture specification
Prevalidation Methodology

- Early evaluation exposes system weaknesses
- Reliability and performance analysis versus staffing level unresolved
- Methodology allows assessment of cost and technical risk
- Seems to mirror Japanese staffing concept

![Graph showing Manpower over Time for United States and Japanese]
Prevalidation methodology

- Control function specification
- System specification
- Design and model system
- Design and model subsystems
- Subsystem specifications
- Design and model components
- Component specifications
- Build components
- Test components
- Integrates components
- Subsystems
- Integrates subsystems
- Components
- Test subsystems
- Design and Validation Phases
- System
- Test system
Building-Block Considerations
Contractor/Subcontractor Relationships

- Requires different approach to subcontractors
- Need to develop:
  - Functional specification
  - Reliability attributes
  - Performance attributes
- Requirements only will not suffice
- Subtleties of building-block interrelationships important
Building-Block Considerations
Contractor/Subcontractor Relationships
(continued)

- Enforcement of rigor on the vendors
- Do we need a two-step procedure with vendors—
  - During building-block definition
  - During hardware/software bid on system
Methodology

Incomplete

- Additional tools
  - Maintainability
  - Availability
  - Survivability
  - Cost
- Software
  - Tie in to top-level system design
- Relationship between full nonlinear simulation and performance model
- Hardware and software build—subsystem validation and verification
- Lab testing
- Flight testing
Tools

Major effort on

- Model development—candidate architecture definition
  - How system works
  - Brief, concise, easy to generate
  - Must include redundancy management operation
- Output data interpretation
  - Complex
  - Very time consuming
Performance Modeling

- Difficult to simulate
  - Conceptual problem
  - Difficult to implement
  - I/O system service
- Detail of simulation is based upon judgement
- Simulation can validate system architecture
- Verification of model with architecture description
- Simulation used through life cycle
- Unexpected insight via performance simulation
Reliability Modeling

Methodology goal: rapid evaluation of architecture alternatives

- Current evaluation cycle too slow
- Tools available for ultrareliable systems
  - Short-duration safety
- Long-duration reliability also important
  - Operation with failures
- Common evaluation tool and similar models (safety, mission, etc.) desirable (mandatory?)
- Level of detail and model simplification currently an art
  - Strong pressure toward small and simple models
  - ASSIST/SURE supports techniques for short-duration problems (long-duration?)
Advanced Information Processing System

(AIPS)

Designed By

Charles Stark Draper Laboratory
AIPS Proof-of-Concept Configuration

FROM: CSDL
FTP HW And SW Provide Failure Protection
I/O Network Elements

- Nodes
- Device interface units (DIU)
- Sensor/Actuator
Flight Control Computer With I/O Connections

Legend:
- Node only
- DIU = Sensors actuators
Architecture
Reference Configuration Overview
Testing Experience

- Application was quickly integrated into complex fault-tolerant AIPS system
  - AIPS simplex application programming model
  - CSDL staff assistance
- Impossible to meet goal of testing system with real time performance demands
  - Slow time testing focused on system level interactions
- Nonintrusive measurements likely requirement for validation
  - During real-time operation with full workload
  - System services or operating system functions critical
  - Not provided for in original AIPS testing/validation concept
General Observations Architecture

AIPS
General Observations
Architecture

- Integrated flight control/propulsion control—feasible
  - Obstacles—mind set problem
- Minimum use of sensors/activators
- Allows for optimum control
- Allows for function migration
- Growth potential
- Subset of Vehicle Management System
General Observations (Cont)
AIPS

Very innovative for its time
• Supports true distributed system
• System redundancy transparent to user
• General set of building blocks—user selected
• Fault containment regions
General Observations (Cont)

AIPS

- Advantages
  - Building blocks allow expansion with minimum change
  - Building block concept supports common hardware/software throughout the airplane
  - Prevalidated building blocks for both hardware and software
  - Ability to mix elements with different reliability requirements
  - Distributed computing possible
  - Function migration possible
  - Minimizes maintenance and logistic issues
General Observations (Cont)  
AIPS

- Advantages (cont)
  - System redundancy is inherent in AIPS design
  - Fault containment region is inherent in AIPS design
  - Pre-emptive priority allows application flexibility
  - Communications protocol allows design for minimum sensor/actuator time skew
  - Concept supports dispatch with failures (need faster network repair time)
  - Variation of components within FTP channel (CP/IOP or CP)
General Observations (Cont)
AIPS

- Concepts needing attention
  - Insufficient documentation
  - IOP/Data Exchange bottleneck
  - IC network traffic uncertain (not modeled)
  - No discernable difference between network and bus for IAPSA requirements
General Observations (Cont)
AIPS

• Concepts needing attention (cont)
  • Complex validation issues
    — IO system services
    — IC system services
    — Pre-emptive priority scheduling
  • Resynchronization of channel during flight not possible with present design
  • System design guidelines not established
  • If IC modeled—it appears system would not work with present timing and loading requirements
Vehicle Management System

- All flight critical functions
- Failure causes loss of aircraft

- Near term - military
- Long term - commercial
Generic VMS Architecture
Photonics used for

- Bus
- Sensors
- I/O
- Actuators
- Computers (20 years)
Benefits of VMS

- **Performance**
  - Unified environment - coordination of all tasks

- **Growth capability**
  - Additional nodes - minimum impact
  - Life cycle replacement - minimum topology impact

- **Reliability**
  - Minimum set of building blocks
  - Minimum part count
  - Common I/O
  - Sharing of sensor data
  - Common redundancy management
A Compact Highly Reliable Avionic Processor

- Multichip Module
- Optical Components
- Low-Power Radiation Hard
- Very Large Scale Integration
- GaAs Circuit
What does all this mean in terms of validation and verification?

- Design for validation is a critical technology
- Need indepth V&V concurrent with design analysis
Solution to V&V

Formal Verification - viable solution to the V&V problem for

- Requirements/Specifications
- Hardware
- Software
- System
Where are we in Formal Verification?

- the following 3 days should tell us!!
MAFT:
The Multicomputer Architecture for Fault-Tolerance

R. M. KIECKHAFER

Computer Science and Engineering
University of Nebraska – Lincoln
Lincoln, NE 68588–0115
(402) 472-2402
rogerk@fergvax.unl.edu

MAFT is a product of the Allied-Signal Aerospace Company, Columbia MD.
Abstract

This presentation discusses several design decisions made and lessons learned in the design of the Multicomputer Architecture for Fault-Tolerance (MAFT). MAFT is a loosely coupled multiprocessor system designed to achieve an unreliability of less than $10^{-10}/hr$ in flight-critical real-time applications.

The presentation begins with an overview of the MAFT design objectives and architecture. It then addresses the fault-tolerant implementation of major system functions in MAFT, including Communication, Task Scheduling, Reconfiguration, Clock Synchronization, Data Handling and Voting, and Error Handling and Recovery.

Special attention is given to the need for Byzantine Agreement or Approximate Agreement in various functions. Different methods were selected to achieve agreement in various subsystems. These methods are illustrated by a more detailed description of the Task Scheduling and Error Handling subsystems.
Presentation Overview

• INTRODUCTION

• SYSTEM FUNCTIONS
  - Communication
  - Task Scheduling
  - Task Reconfiguration
  - Clock Synchronization
  - Data Handling and Voting
  - Error Handling and Recovery

• SUMMARY
Design Objectives

- **RELIABILITY** – $1.0 \times 10^{-9}$ over 10 hours.

- **PERFORMANCE**
  
  200 Hz. – Max Task Iteration Rate
  
  5.5 MIPS – Max Computational Capacity
  
  1.0 MBPS – Max I/O Transfer Rate
  
  5.0 ms. – Min Transport Lag (Input → Output)

- **REUSABLE**
  
  - Functional Partitioning
    
    • Application Specific Functions
    
    • Standard Executive Functions

- **LOW EXECUTIVE OVERHEAD**
  
  - Physical Partitioning
    
    • Separate Executive Processor
    
    • Hardware Intensive
Loosely-Coupled Multiprocessor

- Node \(\Rightarrow\) Processor and Private Memory
- No Shared Memory
- Message-Based Inter-Node Communication
- Common Operating System
MAFT System Architecture

- **OC ⇒ Operations Controller:**
  Special Purpose Device Common to All MAFT Systems

- **AP ⇒ Application Processor:**
  General Purpose Application-Specific Processor
Operations Controller Block Diagram

INTER-NODE MESSAGES IN

RECEIVERS (8)

MESSAGE CHECKER

FAULT TOLERATOR

VOTER

DATA MEMORY

INTER-NODE MESSAGES OUT

TRANSMITTER

SYNCHRONIZER

SCHEDULER

TASK COMMUNICATOR

APPLICATION PROCESSOR
COMMUNICATION
INTER-PROCESSOR COMMUNICATIONS

PRIVATE BROADCAST BUS

- OC
- OC
- OC
- OC
- OC

AP

I/O DEV

- INTRA-NETWORK COMMUNICATION
- MESSAGES TRANSMITTED ON PRIVATE SERIAL BROADCAST BUSSES
- ALL NODES RECEIVE, CHECK AND PROCESS ALL MESSAGES
- MESSAGE TYPES
  - DATA (8/16/32b INT OR BOOL, IEEE STD 32b FLOAT)
  - TASK COMPLETED / STARTED / BRANCH
  - SYNCHRONIZATION / BRANCH INTERACTIVE CONSISTENCY
  - ERROR REPORT

- OC / AP COMMUNICATION
- 16 BIT ASYNCHRONOUS P.I.O. INTERFACE
- LOOKS LIKE "JUST ANOTHER I/O PORT" TO AP
- COMPATIBLE W/ EXISTING UNIPROCESSOR OPER SYST

FEBRUARY 28, 1986
Message Handling

- **TRANSMITTER**
  - Format Msg – NID, Msg Type, Framing, ECC
  - Broadcast Msg

- **RECEIVERS** - 1 per incoming link
  - Accept Properly Framed Bytes
  - Buffer Byte for Message Checker

- **MESSAGE CHECKER**
  - Poll Receivers – 6.4 μs cycle
  - Physical and Logical Checks
  - Steer Good Messages to Other Subsystems
  - Dump Bad Messages into “Bit-Bucket”
LOCAL AP/OC INTERFACE OPERATIONS

1. TASK SWITCHING PROCESS
   - AP: DONE WITH LAST TASK, WHAT IS THE TASK IDENTIFICATION (TID) NUMBER OF THE NEXT TASK.
   - OC: HERE IT IS

2. TRANSFER DATA FROM OC TO AP
   - AP: GIVE ME THE NEXT INPUT DATA VALUE
   - OC: HERE IT IS

3. TRANSFER DATA FROM AP TO OC
   - AP: HERE'S THE NEXT OUTPUT DATA VALUE
   - OC: I GOT IT

+ ATC/RMK FEBRUARY 28, 1986
Typical Task System
PERFORMANCE ISSUES

- STRICTLY PERIODIC SCHEDULER
  - Fast – Freq Well Above Spec – 500 Hz. vs. 200 Hz.
  - Simple – Binary Freq Dist \( f_i = 2^{-i}f_0 \)
  - Flexible – Conditional Branching
  - Efficient – Don’t Keep AP Waiting

- NON-PREEMPTIVE
  - Scheduler Complexity
  - Context Switching Time – Unknown Funct of AP
  - High Frequencies – Short Tasks

- NO OC INTERRUPTS – I/O
  - Scheduler Complexity
  - Predictability
  - High Frequencies – Polling
  - DMA or IOP access to AP Memory
O.C. View of a Task

- INTERNAL FUNCTION IS BLACK BOX

- VISIBLE PROPERTIES OF A TASK
  - Priority (static, unique)
  - Iteration Period
  - Precedence Constraints
  - Min and Max duration Limits
  - Fixed Input and Output Shared Data Sets
  - Branch Condition (asserted at completion)
FAULT-TOLERANCE ISSUES – I

• VARIABLE MODULAR REDUNDANCY
  - Specify Redundancy of Each Individual Task
  - Redundancy Matches Criticality
  - No More Copies Than Necessary

• GLOBAL VERIFICATION
  - Consensus Defines Correctness
  - All Functions Observable and Predictable
  - Replicated Global Scheduler
  - Completed/Started (CS) Message:
    - Node I.D.
    - Started Task I.D.
    - Branch Condition
Message Passing Robustness

- Delivery NOT GUARANTEED

- Single Msg Error Detect. NOT GUARANTEED
  \[ \text{ECC coverage} \geq (1 - 1 \times 10^{-6}) \text{ per msg} \]

- Repeated Undet. Errors PROBABILISTICALLY PRECLUDED
TASK SCHEDULING
FAULT-TOLERANCE ISSUES – II

- DISSIMILARITY BETWEEN COPIES
  - Dissimilar Software and Hardware
    - Guards Against Generic Faults
    - No Guarantee – Knight, Levenson, St. Jean
    - Best Chance of Detecting Error
    - Only Chance of Masking Error
  - Implications
    - Different Numerical Results
    - Different Execution Times
  - Impact on Scheduler
    - Min and Max Execution Time Limits
    - Vote on Branch Conditions in CS Messages
FAULT-TOLERANCE ISSUES – III

• BYZANTINE AGREEMENT

- Definition
  - Agreement on All Messages
  - Validity of Agreement

- Necessity in MAFT
  - Consensus Defines Correctness
  - Must Have Single Consensus

- Preconditions for Disagreement
  - Initial Disagreement – Enhanced by Dissimilarity
  - Assymetric Communication – Minimized by Busses

- Solution – Interactive Consistency (Pease et al.)
  - Global Receipt of All Messages
  - Periodic Synchronized Re-Broadcast Rounds
  - Vote on Received Re-Broadcasts
  - Use Voted Values For All Scheduling Decisions
IMPACT OF FAULT-TOLERANCE

• ALL COPIES DONE BEFORE SUCCESSORS RELEASED

• MAX EXECUTION TIMERS – ASSURE PROGRESS

• CONFIRMATION DELAY – MEAN 2.5 SUB.
  - Only Affects Successors
  - Efficiency Requires Parallel Paths

• FAULT-TOLERANCE LEVELS
  - Single Asymmetric (Byzantine) Fault
  - Double Symmetric Fault
  - Reliability Modelling – $10^{-10}/hr$ with 5 Nodes
# MAFT Timing Hierarchy

<table>
<thead>
<tr>
<th>PERIOD</th>
<th>SPEC</th>
<th>DEFINITION</th>
<th>BOUNDARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB-ATOMIC</td>
<td>Min 400μs</td>
<td>I.C. Rebroadcast Period</td>
<td>Task Inter. Cons. (TIC) Message</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Min Guaranteed Task Duration</td>
<td></td>
</tr>
<tr>
<td>ATOMIC</td>
<td>Min 2–2.8 ms</td>
<td>Highest Freq. Task</td>
<td>System State (SS) Message</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clock Sync. Period</td>
<td></td>
</tr>
<tr>
<td>MASTER</td>
<td>Max 1K Atom. Per.</td>
<td>Lowest Freq. Task</td>
<td>System State (SS) Message</td>
</tr>
</tbody>
</table>
Scheduling Stability Problem

- **SCHEDULING INSTABILITY** – Anomalous or unpredictable variations in total execution time (Makespan) due to variations in system parameters.

- **MULTIPROCESSOR ANOMALIES** – Observation that Makespan can be increased by:
  - Increasing Number of Processors,
  - Relaxing Precedence Constraints,
  - Decreasing Individual Task Durations.

- **DYNAMIC FAILURE** – Condition where all tasks execute properly except that deadlines are missed.
  - Can occur in a fault-free system,
  - Can be induced by instability.
Sample Task System
Instability of Sample Task System

- **STANDARD GANTT CHART** (max task durations)

<table>
<thead>
<tr>
<th>PROCUREMENT 1</th>
<th>T₁</th>
<th>T₂</th>
<th>T₄</th>
<th>T₇</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCUREMENT 2</td>
<td>...</td>
<td>T₃</td>
<td>T₅</td>
<td>T₆</td>
<td>T₈</td>
</tr>
</tbody>
</table>

- **NON-STANDARD GANTT CHART** (shorten T₃ by ε)

<table>
<thead>
<tr>
<th>PROCUREMENT 1</th>
<th>T₁</th>
<th>T₂</th>
<th>T₄</th>
<th>T₈</th>
<th>... ... ...</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCUREMENT 2</td>
<td>...</td>
<td>T₃</td>
<td>T₆</td>
<td>T₅</td>
<td>T₇</td>
<td></td>
</tr>
</tbody>
</table>

- **WHAT HAPPENED?**
  - T₃ finished before T₂,
  - T₆ "ready" before T₅,
  - T₅ displaced by T₆ \(\Rightarrow\) Priority Inversion,
  - Critical path \((T₂ \rightarrow T₇)\) impeded.
Previous Work

• GRAHAM (1969) – Bound Magnitude of Instability

\[
\frac{\omega'}{\omega} = 2 - \frac{1}{N}
\]

- \( \omega = \) Makespan of Standard Gantt Chart,
- \( \omega' = \) Makespan of worst-case schedule,
- \( N = \) Number of Processors.

• MANACHER (1967) – Stabilization Algorithm

- Necessary Pre-conditions
  i. \( \exists \) “fork” in Precedence Graph,
  ii. Successors of forking task run in parallel on Standard Gantt Chart,
  iii. Possible priority inversion around fork.

- Solution – Impose Artificial Dependency around fork.
Stabilized Task System

- MANACHER ARTIFICIAL DEPENDENCY ($T_2 \rightarrow T_6$)

- EFFECT
  - $T_2$ is common parent for both $T_5$ and $T_6$.
  - $T_6$ will be "ready" no earlier than $T_5$.
  - $T_5$ precedes $T_6$ in priority list.
  - $T_6$ cannot be selected before $T_5$. 
Limitations of Manacher’s Solution

- Sufficient, but not always necessary
- Adds Scheduling Overhead (resolve edge)
- Unrealistic System Model
  - Assumes no scheduler overhead,
  - Assumes dynamic allocation,
  - Allows for no Confirmation Delay,
  - Ignores minimum duration bounds,
  - Does not predict magnitude of instability.
Current Research

- Find Necessary and Sufficient Stability Conditions.

- Develop Stabilization Strategies
  - Task System Stabilization
    - Edge Stabilization (Manacher)
    - Vertex Stabilization
    - Hybrid Stabilization
  - Run-Time Scheduler Stabilization
    - Limited Scan Depth
  - Scheduling Algorithm Stabilization
    - Sched. Algorithm Assigns Priorities
    - Constrain to Preclude Necessary Conditions

- Extend System Environment
  - Scheduler Overhead
  - Static Allocation
  - Confirmation Delay
  - Minimum Duration Bounds
SYNCHRONIZATION
MAFT Synchronization

- Periodically Exchange System State (SS) Msgs
  - SS Msg $\Rightarrow$ "Atomic Period" Boundary
  - Synchronization Period = 2 Atomic Periods

- Loosely Synchronized Individual Clocks
  - Msg Exchange $\Rightarrow$ No Separate Clock Lines
  - Physical Separation $\Rightarrow$ Damage Tolerance
  - Robustness to "Common Upset" events

- Synchronization Modes
  - Steady State – Maintain Existing Synchronization
  - Warm Start – Converge to Existing Operating Set
  - Cold Start – Form Initial Operating Set
    - Interactive Convergence to synchronize
    - Interactive Consistency $\Rightarrow$ Steady State
    - Origin of Two-phase algorithm
DATA HANDLING AND VOTING
Typical Sync. Values

- $\epsilon = 7 \text{ } \mu\text{sec} - 600 \text{ ft. separation}$
- $\rho = 5 \cdot 10^{-5}$
- $R = 20 \text{ } m\text{sec} \Rightarrow 10 \text{ } m\text{sec} \text{ Atomic Pd.} \Rightarrow 100 \text{ Hz.}$
- $\rho R = 1 \text{ } \mu\text{sec}$

- No Faults: Max $\delta = 8.5\mu \text{sec}$
- With Faults: Max $\delta = 16.5\mu \text{sec}$
Data Management

• DATA GENERATED BY AP

• BROADCAST IN DATA MESSAGE

• RECEIVED AND PROCESSED BY ALL NDOES
  - Static Limit Check
  - On-The-Fly Vote
  - Dynamic Deviance Check
On-The-Fly Voting I

- Triggered by data message arrival

- Data ID acts as unique variable name

- Use all previous copies of same data ID
  - MS or MME (programmer selectable)
    - Sort serially – high-order-bit first
    - Select 2 "medial" values
    - Average (add and shift)
  - No I.C. vote for boolean types
    - Difficult to implement round 2
    - Usually control data for mode switch
    - ∃ Better way for mode switch
On-The-Fly Voting II

- **DEVIANE CHECK**
  - Compare Each Copy to Voted Value
  - Excessive Difference $\Rightarrow$ error
  - Programmer Sets Limits
  - Generate Error Vector $\Rightarrow$ Source Nodes

- **TERMINATE**
  - Scheduler Says All Copies Done
  - Send Error Vector to Fault-Tolerator
  - Send Voted Value to Data Memory
  - Swap On-line/Off-line Buffers in Data Memory
  - Clear Previously Received Copies from Voter
ERROR HANDLING AND RECOVERY
Fault Classifications

• **BYZANTINE (MALICIOUS)**

  \[-N \geq 3t + 1\]
  \[-r \geq t\]

  Pease et al. (1982)

• **MALICIOUS \(\cup\) BENIGN (self-evident)**

  \[-t = m + b\]
  \[-N \geq 3m + b + 1\]
  \[-r \geq m\]

  Meyer and Pradhan (1987)

• **(ASYMMETRIC \(\cup\) SYMMETRIC) \(\cup\) BENIGN**

  \[-t = a + s + b\]
  \[-N \geq 3a + 2s + b + r + 1\]
  \[-r \geq a\]

  Thambidurai and Park (1989)
Fault Classes by Source

- Can Estimate Separate $\lambda$'s
  - $\lambda_{\text{asy m}} \approx 10^{-6}$
  - $\lambda_{\text{sym}} \approx 10^{-3} \ldots 10^{-4}$

- Generic Fault = Multiple Symmetric
  - $\lambda_{\text{gen}} \approx 10^{-5}$ ?
Error Detection

• Errors Are Manifested In Messages
  - Physical: ECC, framing, length
  - Contents: values
  - Timing or sequencing
  - Existence or non-existence

• Log Errors Over One Atomic Period
  - Errors reported by all subsystems
  - Fault-Tolerator records errors
  - ∃ 31 separate error "flags"
  - ∃ Unique "Penalty Weight" $PW$ for each flag
  - ∃ "Incremental Penalty Count" $IPC$ for each node
  - FOR each flag $f$ reported against node $i$:
    $IPC(i) := IPC(i) + PW(f)$
Error Reporting

• Broadcast ERR(i) Message
  - At beginning of next Atomic Period
  - Contents:
    - $IPC(i)$
    - $BPC(i)$ – Base (current) penalty count
    - All Error Flags for node $i$

• No ERR Message $\Rightarrow$ No Detections
BPC Manipulation

• BPC ⇒ Health Of Node

• Increasing BPC – ERR Message Vote
  - Vote on $BPC(i)$
  - Vote on $IPC(i)$
  - $BPC(i) := BPC(i) + IPC(i)$

• Decreasing BPC – Fixed decrement
  - $\exists$ Penalty Decrement value $PD$
  - At New Master Period
  - $BPC(\ast) := BPC(\ast) - PD$
  - Allows For Eventual Readmission
Exclusion/Readmission

• Recommend Exclusion/Readmission
  - ∃ Exclusion Threshold $T_{excl}$
  - ∃ Admission Threshold $T_{adm}$
  - Recommend in next SS message:
    - $BPC(i) \geq T_{excl} \Rightarrow$ Exclude $i$
    - $BPC(i) \leq T_{adm} \Rightarrow$ Readmit $i$
    - $T_{adm} < BPC(i) < T_{excl} \Rightarrow$ No Change

• I.C. Vote on Recommendations
  - Consistent System State is Critical
  - Free (needed for cold-start)
  - Highly Degraded Systems
  - Common Mode Upset Recovery
ERROR HANDLING (SIMPLEX I.C.)

- Faulty Land
- Error detected
- I.C. Vote on ACC.
- "TRIAL"
- I.C. Vote on EXCL/INCL.
- "SENTENCING"

Error in MSG (I.C. Round 0)
Err MSG's "ACCUSATIONS" (I.C. Round 1)
SS MSG's "VERDICTS" EXCL/INCL (Another I.C. Round 1)
• AP – Diagnostics in Workload

• OC – System Level Self-Test

  - Errors Very Rare
  - Inject Faults to Exercise Error Detection

    • Special self-test Task ID
    • Suspend normal Transmitter Ops
    • Transmit string from self-test ROM
    • Can transmit ANY test scenario

  - Test Results Based On

    • False/Missed Accusations
    • Cyclic Link Check

  - Independent of Actual Bit-Stream
  - Rotate "Originator" Duty
  - Complete Coverage If ANY One Node Correct
Version Management

- **SSV** = System State Vec – eg (2,1,1)
- **VMV** = Version Management Vec – eg (1,1,1)
- **WMV** = Workload Management Vec – (SSV) or (VMV)

**Vectors Used By Different Subsystems**

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Vector</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Voter</td>
<td>VMV</td>
<td>Inactive Copy Ignored For Vote</td>
</tr>
<tr>
<td>Dev Checker</td>
<td>SSV</td>
<td>Inactive Copy Still Monitored</td>
</tr>
<tr>
<td>Scheduler</td>
<td>WMV</td>
<td>Inactive Copy May Not Run</td>
</tr>
</tbody>
</table>

- **WMV = SSV**
  - Inactive Copy Still Executing
  - Actual Tasks Being Monitored
  - Best for Generic Fault Detection

- **WMV = VMV**
  - Inactive Copy Doing Something Else
  - Will Not Be Affected By Generic
  - Can Activate To Replace Sibling
  - Best For Generic Recovery
Synchronizer Error Detection

- MAFT error detection is by consensus
  - Each node reports errors on all nodes.
  - Majority vote confirms or denies accusations.
  - Disagreement with majority may itself be an error.

- Faulty node must be detected by majority of nodes
  - Must be "far enough" out of sync
  - There exists a region of ambiguity
  - Defines size of "Sync Window"
Synchronizer Error Windows

\[ W_s = 5\epsilon + 5\rho R \]
\[ W_h = 11\epsilon + 10\rho R \]

- **\( W_s \) = SOFT ERROR WINDOW**
  - Spans Range of Receipts from Non-Faulty Nodes
  - Error May Not Be Confirmed
  - Inherent Ambiguity
  - Must Suspend Error Disagreement Penalties

- **\( W_h \) = HARD ERROR WINDOW**
  - IF Any non-faulty node detects a Hard-Error
    THEN All non-faulty nodes detect an Error
  - Can demand Corroboration
Typical Sync. Window Values

• \( \epsilon = 7 \ \mu sec - 600 \text{ ft. separation} \)
• \( \rho = 5 \cdot 10^{-5} \)
• \( R = 20 \text{ msec} \Rightarrow 10 \text{ msec Atomic Pd.} \Rightarrow 100 \text{ Hz.} \)
• \( \rho R = 1 \ \mu sec \)

• No Faults: \( \text{Max } \delta = 8.5 \mu sec \)
• With Faults: \( \text{Max } \delta = 16.5 \mu sec \)

• \( W_s = 40 \mu sec \)
• \( W_h = 87 \mu sec \)
SUMMARY COMMENTS ON THE APPLICATION OF MAFT TECHNOLOGY

1. CAPABILITIES

- BASIS OF A GENERIC REAL-TIME MULTICOMPUTER SYSTEM
- REMOVES F.T. OVERHEAD FROM APPLICATION PROCESSOR
- HANDLES ALL REDUNDANCY MANAGEMENT WITHIN COMPUTER
- ASSISTS IN REDUNDANCY MANAGEMENT OF I/O SYSTEM

2. FLEXIBILITY

- INDEPENDENT OF I/O ARCHITECTURE
- HIGHLY RECONFIGURABLE AND GRACEFULLY DEGRADABLE
- PROVIDES MECHANISMS, NOT POLICIES

3. USABILITY

MARCH 19, 1985
ADVANTAGES OF APPROACH

- Partitioned approach significantly reduces processor overhead
- Data driven architecture much faster than software implementation
- Not dependent upon architecture of application processor
- Redundancy is "task-based" and flexible
- Suitable for high reliability and high performance applications
MAFT Bibliography

References


Design For Validation
Based on Formal Methods

Ricky W. Butler
NASA Langley Research Center
Hampton, VA 23665

N91-17564
VALIDATION OF ULTRA-RELIABLE SYSTEMS

DECOMPOSES INTO TWO SUBPROBLEMS

1. Quantification of probability of system failure due to physical failure
2. Establishing that DESIGN ERRORS are not present

\[1\text{(note. Quantification of 2 is infeasible)}\]
Achieving Ultra-Reliable Software
(Approaches)

• Testing (Lots of it)

• Design Diversity (e.g. N-version programming)

• 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

1. Separate Design/Implementation Teams
2. Same Specification
3. Multiple Versions
4. Non-exact Threshold Voters
5. Hope design flaws manifest errors independently or nearly so.
The Big Problem For Design Diversity Advocates

- experiments in the low-nominal reliability region have shown that the number of near-coincident failures far exceeds the number predicted by an independence model.

- Certainly independence cannot be assumed axiomatically for ultrareliable regime

- If cannot assume independence must measure correlations.
Quantification of N-version programs not feasible in the ultrareliable regime

- Since one cannot assume independence, it must be treated as black box

- Back to life-testing problem again

- Any alternative model would have to be validated. But How?
How do we get ultra-reliable numbers for hardware (physical failure)?

THE ONLY THING THAT ENABLES QUANTIFICATION OF ULTRA-RELIABILITY FOR H/W IS THE

INDEPENDENCE ASSUMPTION !!

- THE INDEPENDENCE ASSUMPTION CANNOT BE DEMONSTRATED FOR MULTI-VERSION S/W IN THE ULTRA-RELIABLE REGION
The Danger of Design Diversity (N-version Programming, Recovery Blocks, etc.)

- creates an "illusion" of ultra-reliability. By assuming independence, the advocates of S/W fault-tolerance generate ultra-high estimates of reliability.

- As long as industry/certification agencies believe that S/W fault-tolerance will solve the problem, formal methods will not be pursued.
Design For Validation

1. Designing a system in a manner that a complete and accurate reliability model can be constructed. All parameters of the model which 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 which minimize the number of measurable parameters in order to reduce the validation cost. A design which has exceptional performance properties yet requires the measurement of hundreds of parameters (e.g. say by time-consuming fault-injection experiments) would be rejected over a less capable system that requires minimal experimentation.

3. The system is designed in a manner that enables a proof of correctness of its logical structure. Thus, the reliability model does not include transitions representing design errors.

4. The reliability model is shown to be accurate with respect to the system implementation. This is accomplished analytically.
Illustration 1

Suppose we must design a simple fault-tolerant system with a probability of failure no greater than $2 \times 10^{-6}$ whose maximum mission time is 10 hours.

- We quickly eliminate the use of a simplex processor since there is no technology which can produce a processor with this low of a failure rate.

- Thus, we begin to explore the notion of fault-tolerance. We next consider the use of redundancy—how about a dual? When the first processor fails, we will automatically switch to the other processor.

Unfortunately, our design suffers from one major problem. It is impossible to prove that any implementation behaves in accordance with this model.

The problem is that one cannot design a dual system which can detect the failure of the first processor and switch to the second 100% of the time. Thus, we must accept the fact that there is a single-point failure in our system and include it in our reliability model.
SURE Run

Now we have a parameter in our model which must be measured—C. It represents the fraction of single faults from which the system successfully recovers. Can this parameter can be measured in a feasible amount of time (i.e. say less than year) with statistical significance?

```
$ sure
SURE V7.5  NASA Langley Research Center
1? read dualspf
2:
3:  LAMBDA = 1E-4;
4:  C = .9 TO 1 BY .01;
5:  1,2 = 2*LAMBDA*C;
6:  2,3 = LAMBDA;
7:  1,4 = 2*(1-C)*LAMBDA;
8? run

<table>
<thead>
<tr>
<th>C</th>
<th>LOWERBOUND</th>
<th>UPPERBOUND</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.00000e-01</td>
<td>2.00699e-04</td>
<td>2.00900e-04</td>
</tr>
<tr>
<td>9.10000e-01</td>
<td>1.80729e-04</td>
<td>1.80910e-04</td>
</tr>
<tr>
<td>9.20000e-01</td>
<td>1.60759e-04</td>
<td>1.60920e-04</td>
</tr>
<tr>
<td>9.30000e-01</td>
<td>1.40789e-04</td>
<td>1.40930e-04</td>
</tr>
<tr>
<td>9.40000e-01</td>
<td>1.20819e-04</td>
<td>1.20940e-04</td>
</tr>
<tr>
<td>9.50000e-01</td>
<td>1.00849e-04</td>
<td>1.00950e-04</td>
</tr>
<tr>
<td>9.60000e-01</td>
<td>8.08790e-05</td>
<td>8.09600e-05</td>
</tr>
<tr>
<td>9.70000e-01</td>
<td>6.09090e-05</td>
<td>6.09700e-05</td>
</tr>
<tr>
<td>9.80000e-01</td>
<td>4.09390e-05</td>
<td>4.09800e-05</td>
</tr>
<tr>
<td>9.90000e-01</td>
<td>2.09690e-05</td>
<td>2.09900e-05</td>
</tr>
<tr>
<td>1.00000e+00</td>
<td>9.99000e-07</td>
<td>1.00000e-06</td>
</tr>
</tbody>
</table>
```

From this run we know that C must be between 0.9 and 1.0 in order to meet our reliability goal. We rerun the model to get a closer value:
2nd Run

$ sure
9? read dualspf
10:
11: LAMBDA = 1E-4;
12: C = .999 to 1 by 0.0001;
13: 1.2 = 2*LAMBDA*C;
14: 2.3 = LAMBDA;
15: 1.4 = 2*(1-C)*LAMBDA;

0.02 SECS. TO READ MODEL FILE
16? run

<table>
<thead>
<tr>
<th>C</th>
<th>LOWERBOUND</th>
<th>UPPERBOUND</th>
<th>COMMENTS</th>
<th>RUN #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.99000e-01</td>
<td>2.99600e-06</td>
<td>2.99900e-06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.99100e-01</td>
<td>2.79630e-06</td>
<td>2.79910e-06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.99200e-01</td>
<td>2.59660e-06</td>
<td>2.59920e-06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.99300e-01</td>
<td>2.39690e-06</td>
<td>2.39930e-06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.99400e-01</td>
<td>2.19720e-06</td>
<td>2.19940e-06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.99500e-01</td>
<td>1.99750e-06</td>
<td>1.99950e-06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.99600e-01</td>
<td>1.79780e-06</td>
<td>1.79960e-06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.99700e-01</td>
<td>1.59810e-06</td>
<td>1.59970e-06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.99800e-01</td>
<td>1.39840e-06</td>
<td>1.39980e-06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.99900e-01</td>
<td>1.19870e-06</td>
<td>1.19990e-06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.00000e+00</td>
<td>9.99000e-07</td>
<td>1.00000e-06</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Now, we can see that we must demonstrate that C is greater than 0.9995.
- It can be shown that 20000 observations are necessary to estimate this parameter to a reasonable level of statistical significance.
- If we optimistically assume that each fault injection requires 1 minute, then this validation exercise would require 330 hours (i.e. 14 days).

In this case, we decide we can live with this amount of testing and proceed to develop our system.
Designing System with Much Higher Reliability

Now suppose we want to meet the reliability goal of $1 - 10^{-9}$.

We decide to use a nonreconfigurable 5-plex (5MR) and build a processor with a failure rate of $10^{-5}$/hour. The probability of system failure is plotted as a function of $1-C$:

- The value of $C$ must now be greater than 0.9999982.
- It can be shown that over a million fault injections would be required to measure this parameter even if we are very optimistic about the testing process.
- If each injection required 1 minute, this would require almost 1.9 years of non-stop fault injections.
A better Way—via Design For Validation

It would be nice if we could design our system so that such an experiment is unnecessary.

- The system is designed so that a single point failure cannot cause system failure (i.e. $C = 1$).
- This is demonstrated to be true by formal proof.
- Thus, one uses the power of analysis to eliminate fault-injection style testing.
WHY FORMAL METHODS?

The successful engineering of complex computing systems will require the application of *mathematically based analysis* analogous to the structural analysis performed before a bridge or airplane wing is built.
Draft Interim Defence Standard 00-55

Quote from the forward to the Draft Standard:

The Steering Group "has determined that the current approach which is based on system testing and oversight of the design process will, in the long-term, become cumbersome and inefficient for the assurance of the safety of increasingly sophisticated software".

"The Steering Group therefore proposes the adoption of Formal Design Methods, based on rigorous mathematical principles, for the implementation of safety-critical computer software".
Levels of Formal Methods

Level 0: Static Code Analysis (i.e. no semantic analysis)

Level 1: Specification using mathematical logic or language with a formal semantics (i.e. meaning expressible in logic)

Level 2: Formal Specification + Hand Proofs

Level 3: Formal Specification + Mechanical Proofs
Summary

- Testing and design-diversity techniques inadequate.

- A design-for-validation based on formal methods is needed for the digital flight control systems problem.

- Formal methods will play a major role in the development of future high reliability digital systems.
What FM can offer DFCS Design

John Rushby

Computer Science Laboratory
SRI International
Overview

- What has actually gone wrong in practice?
- What is the pattern?
- What is the solution?
Advanced Fighter Technology Integration (AFTI) F16

- Triplex DFCS to provide two-fail operative design

- Analog backup

- Digital computers were not synchronized

- “General Dynamics believed synchronization would introduce a single-point failure caused by EMI and lightning effects”
AFTI F16 DFCS Redundancy Management

- Each computer samples sensors independently, uses average of the good channels, with wide threshold

- Single output channel selected from among the good channels

- Output threshold 15% plus rate of change

- Four bad values in a row and the channel is voted out
AFTI F16 Flight Test, Flight 15

- Stores Management System (SMS) relays pilot requests for mode changes to DFCS
- An unknown failure in the SMS caused it to request mode changes 50 times a second
- DFCS responded at a rate of 5 mode changes per second
- Pilot said aircraft felt like it was in turbulence
- Analysis showed that if aircraft had been maneuvering at the time, DFCS would have failed
AFTI F16 Flight Test, Flight 36

- Control law problem led to "departure" of three seconds duration

- Sideslip exceeded 20°, normal acceleration exceeded −4g, then +7g, angle of attack went to −10°, then +20°, aircraft rolled 360°, vertical tail exceeded design load, failure indications from canard hydraulics, and air data sensor

- Side air data probe blanked by canard at high AOA

- Wide threshold passed error, different channels took different paths through control laws

- Analysis showed this would cause complete failure of DFCS and reversion to analog backup for several areas of flight envelope
AFTI F16 Flight Test, Flight 44

- Asynchronous operation, skew, and sensor noise led each channel to declare the others failed

- Analog backup not selected (simultaneous failure of two channels not anticipated)

- Aircraft flown home on a single digital channel

- No hardware failures had occurred
AFTI F16 Flight Test

- Repeated channel failure indication in flight was traced to roll-axis software switch

- Sensor noise and asynchronous operation caused one channel to take a different path through the control laws

- Decided to vote the software switch

- Extensive simulation and testing performed

- Next flight, same problem still there

- Found that although switch value was voted, the unvoted value was used
X29 Flight Test

• Three sources of air data on X29A: nose and two side probes

• If value from nose is within threshold of both side probes, use nose probe value

• Threshold is large due to position errors in certain flight modes

• If nose probe failed to zero at low speed it would still be within threshold of correct readings

• Aircraft would become unstable and "depart"

• Caught in simulation but 162 flights had been at risk
HiMAT Flight Test

- Single failure in redundant uplink hardware

- Software detected this, and continued operation

- But would not allow the landing skids to be deployed

- Aircraft landed with skid retracted, sustained little damage

- Traced to timing change in the software that had survived extensive testing
Gripen Fight Test, Flight 6

- Unstable aircraft
- Triplex DFCS with Triplex analog backup
- Yaw oscillations observed on several flights
- Final flight had uncontrollable pitch oscillations
- Crashed on landing, broke left main gear, flipped
- Traced to control laws
Space

- Voyager computer clocks skipped 8 seconds at Jupiter due to high radiation levels (AW&ST Aug 7, 1989)

- So "continuous resynchronization" provided at Neptune

- Also, remember STS-1: "The bug heard round the world" (SEN Oct 1981)
FDIR and Crew Interface

- Imaginary crash scenario
- Broken fan blade on port engine
- Port vibration sensor saturates, limiter cuts in
- Vibration travels down wing, shakes starboard engine
- Starboard vibration sensor reports the attenuated vibration
- Only starboard vibration warning light comes on in cockpit
- Pilot shuts down the good engine, crashes short of runway
- Similar to British Midland 737 crash in 1989
Complexity and Integration

• "The FMS of the A320 'was still revealing software bugs until mid-January,' according to Gérard Guyot (Airbus test and development director). There was no particular type of bug in any particular function, he says. 'We just had a lot of flying to do in order to check it all out. Then suddenly it was working,' he says with a grin" (Flight International, 27 Feb 1989)

• The ATF hardware is ready to go, but cannot be flown because the software engineers "can't get all the 0's and 1's in the right order" (Northrop Engineer, 7 Aug, 1990)
## Complexity and Integration

<table>
<thead>
<tr>
<th></th>
<th>A300</th>
<th>A310</th>
<th>A320</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>As of early 1988</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number in service</td>
<td>16</td>
<td>149</td>
<td>3</td>
</tr>
<tr>
<td>Flight Hours</td>
<td>16,000</td>
<td>810,000</td>
<td>2,000</td>
</tr>
</tbody>
</table>

### Computers

- **Autopilot**: 2 FCC
- **Rudder**: 2 FAC
- **Autothrottle**: 1 TCC
- **Slats and flaps**: 1 or 2 TCC
- **Elevator/aileron**: 2 SFCC
- **Spoilers**: 2 EFCU
- **Fuel management**: 2 FLAC
- **Instruments**: 2 CGCC
- **Brakes**: 3 SGU
- **Engines**: 2 FADEC
- **Engines**: 2 FADEC
- **Engines**: 2 FADEC
- **Engines**: 2 FADEC
- **Engines**: 2 FADEC
- **Engines**: 2 FADEC
- **Engines**: 2 FADEC
- **Engines**: 2 FADEC
Analog, Mechanical Backups

- Do mechanical and analog backups reduce the requirement for ultra-reliability in DFCS?

- Not if the DFCS is providing stability augmentation or envelope protection

- Similar problem in ATC—potential to move traffic at higher rates than the backup can handle

- No FAA certification credit for mechanical rudder and trim-tab on A320
Analysis: Dale Mackall, NASA Engineer
AFTI F16 Flight Test

- Nearly all failure indications were not due to actual hardware failures, but to design oversights concerning asynchronous computer operation.

- Failures due to lack of understanding of interactions among
  - Air data system
  - Redundancy management software
  - Flight control laws
FLIGHT CONTROL SYSTEM RELIABILITY HEAVILY DEPENDENT ON SYSTEM INTERACTIONS

CONTROL SYSTEM RELIABILITY

- HARDWARE RELIABILITY
- SOFTWARE RELIABILITY
- SYSTEMS INTERACTIONS
- EXTERNAL EVENTS
Analysis: NASA-LaRC 1988 FCDS Technology Workshop

• Lack of fully effective design and validation methods with support tools to enable engineering of highly-integrated, flight-critical digital systems

• Complexity of failure containment, test coverage, FMEA, redundancy management, especially in the face of increased integration of flight-critical functions

• Sources of failure:
  o Multiple independent faults (never observed)
  o Single point failures (observed sometimes)
  o Domino failures (most common?)
Analysis: Scientific Foundations

- It is time to place the development of real-time systems on a firm scientific basis. Real-time systems are built one way or another because that was the way the last one was built. And, since the last one worked, we hope that the next one will. (Fred Schneider)

- "Not far from there (CNRS-LAAS), Airbus Industries builds the Airbus A320s. These are the first commercial aircraft controlled solely by a fault-tolerant, diverse computing system. Strangely enough this development owes little to academia. (IEEE Micro, April 1989, p6)
Analysis

- The problems of DFCS are the problems of systems whose complexity has exceeded the reach of the intellectual tools employed.

- Intuition, experience, and techniques derived from mechanical and analog systems are insufficient for complex, integrated, digital systems.
Synthesis

- Computer science has been addressing issues of systematic design, fault tolerance, and the mastery of complexity with some (limited) success for the last 20 years.

- But there has been little interest in learning about, and applying this knowledge to, real-time control systems in general (and little opportunity to apply it to DFCS).

- And little of the lore and wisdom of practical real-time control system design has been captured and analyzed.
What Computer Science Can Offer DFCS

- Systematic techniques for the construction of trustworthy software, including:
  - Techniques for the precise specification of requirements and the development of designs
  - Systematic approaches to the design and structuring of distributed and concurrent systems
  - Fault tolerant algorithms
  - Systematic methods of testing and analytic methods of verification

- Where do formal methods come in?
Applied Mathematics and Engineering

• Established engineering disciplines use applied mathematics
  ○ As a *notation* for describing systems
  ○ As an analytical tool for calculating and *predicting* the behavior of systems

• Computers can provide speed and accuracy for the calculations
Applied Mathematics and Software Engineering

• The applied mathematics of software is formal logic

• Formal Logic can provide
  ○ A notation for describing software designs—formal specification
  ○ A calculus for analyzing and predicting the behavior of systems—formal verification

• Computers can provide speed and accuracy for the calculations

• Calculating the behavior of software is an exercise in formal reasoning—i.e., theorem proving
Formal Methods

- Methodologies for using mathematics in software engineering

- Can be applied at many different levels, for both description and analysis

  0. No application of formal methods
  1. Quasi-formal pencil and paper techniques
  2. Mechanized quasi-formal methods
  3. Fully formal pencil and paper techniques
  4. Mechanically checked fully formal techniques
Benefits of Formal Specification

- Unambiguous description facilitates communication among engineers

- Early detection of certain errors

- Encourages systematic, thoughtful approach, reuse of well-understood concepts

- As documentation, reduces some of the difficulties in maintenance and modification
Benefits of Formal Verification

- Subjects the system to extreme scrutiny, increasing designers' understanding of their own creation

- Helps identify assumptions, increases confidence

- Encourages simple, direct designs, austere requirements—better systems

- Encourages and supports a systematic, derivational approach to system design

- Complements testing and allows it to focus on fundamentals
Conclusion: What FM Can Offer DFCS

• Precise *notations* for specifying requirements and designs

• *Concepts* and *structure* for systematic design

• Intellectual tools for *analyzing* the consistency of specifications and the conformance of designs

• A way to regain *intellectual mastery* of complex systems and their interactions
Recommendations

• Just adding formal methods to existing practice is inappropriate

• Capture and analyze lore and wisdom (and mistakes) of actual DFCS designs

• Apply modern Computer Science (including Formal Methods) to develop building blocks for principled DFCS design

• Ultimately, build one and fly it!
What Can Formal Methods Offer to Digital Flight Control Systems Design?

Donald I. Good
Computational Logic, Inc.

Abstract

Formal methods research is beginning to produce methods which will enable mathematical modeling of the physical behavior of digital hardware and software systems. The development of these methods directly supports the NASA mission of increasing the scope and effectiveness of flight system modeling capabilities.

The conventional, continuous mathematics that is used extensively in modeling flight systems is not adequate for accurate modeling of digital systems. Therefore, the current practice of digital flight control system design has not had the benefits of extensive mathematical modeling which are common in other parts of flight system engineering.

Formal methods research is showing that by using discrete mathematics, very accurate modeling of digital systems is possible. These discrete modeling methods are still in an embryonic stage. But when they are fully developed, they will bring the traditional benefits of modeling to digital hardware and software design. Sound reasoning about accurate mathematical models of flight control systems can be an important part of reducing the risks of unsafe flight control.
What Can Formal Methods Offer to Digital Flight Control Systems Design?

Donald I. Good

Computational Logic, Inc.
1717 West Sixth, Suite 290
Austin, Texas 78703

512-322-9951

good@cli.com
"Formal Methods" Enable

Mathematical Modeling

of

Digital Systems

(Hardware and Software)

**NASA Mission Objective:** Increase the scope and effectiveness of flight system modeling capabilities. -- Lee Holcomb, NASA HQ, 1990.
Why Model?

For either design of a new system or operation of an old one, modeling provides...

Benefits: early error detection
- Saves time
- Saves money
- Saves operational disruption
- Saves operational mishaps

Risks: model misrepresents system
- Inaccurate
- Incomplete

Kinds of models: physical, analog, schematic, mathematical.

Why a Mathematical Model?

- High abstraction
- High precision
- Simulate by manipulating symbols
- Represent large classes of system states
- Use mathematical deduction

Get a lot of system simulation for a little symbol manipulation.
Operational Safety

Operating a system safely requires

• accurate predictions
of how it will behave.

Accurate predictions can be obtained from

• sound deductions about
• accurate mathematical models
of system behavior.
A Classic Model

Free Fall Distance:

\[ f(b, t) = \frac{[g(b) \times t^2]}{2} \]

\( g(b) = \begin{cases} 
32 & \text{if } b= \text{"earth"} \\
\text{else if } b= \text{"moon"} & \text{then ...} 
\end{cases} \)

t is time (sec)

\( f(b, t) \) is distance (ft)

Simulation:

\[ f(\text{"earth"}, 0.7) = \frac{[32 \times 0.7^2]}{2} \]
\[ = 16 \times 0.49 \]
\[ = 7.84 \text{ ft} \]
Power of Mathematical Deduction

Suppose 0 le t0 le t1.

\[ t \text{ in } [t0..t1] \]

\[ f("earth", t) \text{ in } (32 * [t0..t1]**2) / 2 \]

\[ f("earth", t) \text{ in } 16 * [t0..t1]**2 \]

\[ f("earth", t) \text{ in } 16 * [t0**2..t1**2] \]

(** is monotonic)

Physical simulation of this result is impossible because [t0..t1] contains an infinite number of values.
Validating a Model

- Ultimately, the accuracy of a model of a physical system must be validated by testing it against measured, observed behavior of the actual physical system.

- One cannot construct a mathematical proof that a model is an accurate representation of a physical system.

- Typically, one iterates through a process of
  - stating a mathematical model
  - testing it against physical observations
  - adjusting the model
Hardware Model Observables

A hardware system is composed of physical switches.


Next page.
The switches of Function Table A are being set by Pte. Homer Speer and Capt. Iwan Goldstein.

The manually set function tables served as read-only memory units in the test. These were School of Electrical Engineering, University of Pennsylvania.

ORIGINAL PAGE
BLACK AND WHITE PHOTOGRAPH

ORIGINAL PAGE IS
OF POOR QUALITY
Use Discrete Mathematics to Model Hardware

- **Switches** by binary digits

- **Operation** by recursive functions

<table>
<thead>
<tr>
<th>s0</th>
<th>0 1 1 0 0 0 0 1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>1 0 1 0 0 1 1 0 0 0 0</td>
</tr>
<tr>
<td>s2</td>
<td>1 1 1 0 0 0 1 0 1 0 1</td>
</tr>
</tbody>
</table>

...
An MC68020 Machine Model

\[
MC68020(s,n) = \\
\text{if } \text{haltp}(s) \text{ or } n=0 \\
\text{then } s \\
\text{else } MC68020(\text{NEXT}(s), n-1)
\]

\[
\text{NEXT}(s) = \\
\text{if } \text{evenp}(\text{pc}(s)) \\
\text{then if } \text{pc\_readp}(\text{mem}(s), \text{pc}(s)) \\
\quad \text{then EXECUTE}(\text{FETCH}(\text{pc}(s), s), \\
\quad \quad \text{update\_pc}(s, \ldots)) \\
\quad \text{else } \text{halt}(s, \text{pc\_signal}) \\
\text{else } \text{halt}(s, \text{pc\_odd\_signal})
\]

\[
\text{EXECUTE}(\text{ins}, s) = \\
\ldots \quad \text{[50 pages for 90\% user ins.]} \quad \ldots
\]

Provides a mathematically precise and consistent machine language reference manual.

The VIPER Machine

A 32-bit microprocessor "whose functions are totally predictable."

- Accumulator
- 2 index registers
- Program counter
- Comparison register
- 16 instructions


A VIPER Machine Model

NEXT(ram, p, a, x, y, b, stop) =
  if stop
  then (ram, p, a, x, y, b, stop)
  else (noinc \/ illegaladdr) \/
       if (illegalcl \/ illegalsp)
           \/ (illegalonp \/ illegalwr)
       then (ram, newp, a, x, y, b, T)
       else ... [about 7 pages] ...

where

ram - a memory of 32-bit words
p  - 20-bit program counter
a  - 32-bit accumulator
x, y - 32-bit index registers
b  - 1 bit compare result register
stop - stop flag
The FM8502 Machine

A 32-bit microprocessor.
• 2 address architecture
• 4 addressing modes
• 8 general purpose registers
• \(2^{19}\) 20-bit instructions


An FM8502 Machine Model

FM8502(ms,mn) =
    if not(listp(mn))
    then ms
    else FM8502(NEXT(ms),
              rest(mn))

NEXT(ms) =
    list(next_memory    (ms),
         next_register_file(ms),
         next_carry_flag    (ms),
         next_overflow_flag(ms),
         next_zero_flag     (ms),
         next_negative_flag(ms) )

... [about 10 pages] ...
An FM8502
Register Transfer Model

GATES(gs, gn) =
  if not(listp(gn))
  then gs
  else GATES(COMB_LOGIC(gs, car(gn)),
             cdr(gn))

COMB_LOGIC(gs, gn) =
... [on bit operators, e.g., b_xor] ...

where

gs - [regs, flags, mem, int-reg]
regs - 8 32-bit vectors
flags - 4 Booleans
mem - 2^32 32-bit vectors
int-reg - 32-bit vectors for internal
          registers, flags, latches
Connecting the Models

Theorem: \( H(\text{ms}, \text{mn}) \rightarrow \text{fm8502(\text{ms}, \text{mn})} = \cup (\text{gates}(\text{D(\text{ms})}, \text{Kg(\text{ms}, \text{mn}, \text{md})})) \)

Under the conditions \( H \),

- the \text{fm8502 model} is just as \textbf{accurate} as \text{gates}
- but with some \textbf{details} \textbf{suppressed} by \( \cup \).
Software Model Observables

Programming languages provide

a wide variety of ways

of describing them, but

the observables are still switches,

and so are programs!
Models of Programmed Machines

- A machine is **programmed** by **setting** the **switches** which it will interpret as instructions during its operation. (Before stored-program machines, this process was called "setting up" the machine.)

```
+-------------------+
| 0 1 1 0 0 0 0 1 1 1 1 |
| prog | data |
+-------------------+
```

- These switches are the **program**. They control the subsequent operation of the machine.
- A **computer program** is a physical **control mechanism**.
- The bit string "011000" is a **mathematical description** of the control mechanism.
A Model of a Programmed Machine

A model of machine $M$ operating on initial state $s_0$ for $k(s_0)$ steps under the control of the program described by $p_0$ is given by

$$M(s_0, k(s_0))$$

where

$s_0$ - a machine state such that $\text{prog}(s_0) = p_0$

$\text{prog}(s)$ - a function that extracts the program description from $s$

Operating Requirements

A model of a machine programmed to satisfy an operating requirement $R(s_0, s_k)$ is given by

$$R(s_0, M(s_0, k(s_0)))$$
A Program Description, p0

[752 16-bit words]

note-82.mss: 22
08/27/90
The Kit Separation Kernel

- Uses a modified FM8501 (ms, mn) machine
- Interrupts for timer and I/O
- Process management
  - fixed number of processes
  - process scheduling (round robin)
  - process communication (message passing)
  - response to error conditions
- Device management for character I/O to asynchronous devices
- Memory management uses hardware protection

The CLInc Stack

\[ \text{o} \rightarrow \text{uGypsy}(yx, yp, yd, yn) \rightarrow o \]
\[ \text{Compile} \rightarrow \text{Young} \rightarrow \text{p\_display} \]
\[ \text{Link-assemble} \rightarrow \text{Moore} \rightarrow \text{m\_display} \]
\[ \text{Reify} \rightarrow \text{Hunt} \rightarrow \text{g\_display} \]
\[ \text{o} \rightarrow \text{piton}(ps, pn) \rightarrow o \]
\[ \text{o} \rightarrow \text{fm8502}(ms, mn) \rightarrow o \]
\[ \text{o} \rightarrow \text{gates}(gs, gn) \rightarrow o \]

The Piton Language

The Piton language has

- execute-only program space
- read/write global arrays
- recursive subroutine calls
- formal parameters
- user-visible stack
- stack-based instructions
- flow-of-control instructions.

The cross assembler produces an FM8502 binary core image.
The Micro Gypsy Language

The Micro Gypsy subset of Gypsy has

- types integer, boolean, character
- one dimensional arrays
- procedure calls with pass by reference parameters
- sequential control structures if, loop,
- condition handling signal..when.

The compiler produces Piton.
The Stack Theorem

Theorem: \( H' (yx, yp, yd, yn) \rightarrow \)
\[ uGypsy(yx, yp, yd, yn) = \]
\[ U' (gates(D' (yx, yp, yd), Kg' (yx, yp, yd, yn, md))) \]

Proof: Mechanically checked.

Under the conditions \( H' \),
- the \( uGypsy \) model is just as accurate as \( gates \)
- but with many details suppressed by \( U' \).

Boyer-Moore Logic


A Hierarchy of Models of a Programmed Machine

\[ R(yx_0, yp_0, yd_0, ydk) \]
\[ uGypsy(yx_0, yp_0, yd_0, yk(yx_0, yp_0, yd_0)) \]
\[ piton(ps_0, pk(ps_0)) \]
\[ fm8502(ms_0, mk(ms_0)) \]
\[ gates(gs_0, gk(gs_0)) \]

Corresponding to these is a hierarchy of program descriptions....
procedure mult(var ans:fm8502_int;
    i,j:fm8502_int) =
begin
  ENTRY j ge 0;
  EXIT ans = NTIMES(i,j);
  pending;
end;

type fm8502_int =
    integer[-(2**31)..(2**31)-1];

{A Simple Problem Domain Theory}

function NTIMES(x,y:integer):integer =
begin
  exit (assume result =
    if y = 0 then 0
    else if y = 1 then x
      else x + NTIMES(x,y-1)
    fi
    fi);
end;
Gypsy Program Description

procedure mult(var ans:fm8502_int;
i,j:fm8502_int) =

begin
ENTRY j ge 0;
EXIT ans = NTIMES(i,j);
var k:fm8502_int := 0;
k := j;
an := 0;
loop
  ASSERT j ge 0 & k in [0..j]
  & ans = NTIMES(i,j-k);
  if k le 0 then leave end;
an := ans + i;
k := k - 1;
end;
end;
Piton Program Description

(MG-MULT
(K ZERO ONE B ANS I J) ;formals
NIL ;locals
(PUSH-LOCAL ANS) ;ans := 0;
(PUSH-CONSTANT (INT 0))
(CALL MG-SIMPLE-CONSTANT-ASSIGNMENT)
(PUSH-LOCAL K) ;k := j;
(PUSH-LOCAL J)
(CALL MG-SIMPLE-VARIABLE-ASSIGNMENT)
(DL L-1 NIL (NO-OP)) ;loop
(PUSH-LOCAL B) ; b := k le 0
(PUSH-LOCAL K)
(PUSH-LOCAL ZERO)
(CALL MG-INTEGER-LE) ; if b then leave
(PUSH-LOCAL B)
(FETCH-TEMP-STK)
(TEST-BOOL-AND-JUMP FALSE L-3)
(PUSH-CONSTANT (NAT 0))
(POP-GLOBAL C-C)
(JUMP L-2)
(JUMP L-4)
(DL L-3 NIL (NO-OP))
(DL L-4 NIL (NO-OP))
(PUSH-LOCAL ANS) ; ans := ans + i;
(PUSH-LOCAL ANS)
(PUSH-LOCAL ANS)
(PUSH-LOCAL I)
(CALL MG-INTEGER-ADD)
(PUSH-GLOBAL C-C)
... [14 more support routines] ...
FM8502 Program Description

(U-STATE)

... (10 more pages) ...

note-82.mss: 33

08/27/90
Mathematical Requirements

- Unambiguous: Requirements have a well-defined interpretation that tells exactly what they do say.

- Analyzable: Do the requirements say the "right" thing?

  \[ R(x, y) \rightarrow \text{good\_thing}(x, y) \]

- Consistency: Requirements contain no contradictions.

- Enable modeling a program component before building it (and thereby save the time and cost of designing a poor program.)

To get these benefits, the requirements notation must have a rigorous mathematical foundation (semantics).
Design >> Requirements

- There is more to designing a digital system than just stating and refining mathematical requirements.
- One must still construct a program for some machine.
- Mathematical models of commonly used languages and machines are still very scarce.
Summary

For either design of a new system or operation of an old one, mathematical modeling of digital flight control systems offers benefits:

Benefits: early error detection
- Saves time
- Saves money
- Saves operational disruption
- Saves operational mishaps

Risks: model misrepresents system
- Inaccurate
- Incomplete

note-82.mss: 36  08/27/90
Conventional Non-Wisdom

Use "formal methods" (mathematical modeling)
- only after a system is built to certify it
- only before a system is built to design it
- to guarantee perfect system behavior
- to eliminate the need for testing
High Level Design Proof of a Reliable Computing Platform

Ben L. Di Vito
Ricky W. Butler
James L. Caldwell

NASA Langley Research Center
Hampton, VA 23665
Outline

- Research Objectives
- Reliable Computing Platform
- High-Level Design Specifications
- Correctness Proofs
- Voting Patterns
Digital Flight Control Systems

Aerodynamic Properties

Continuous Differential Equations Model

Control Law Block Diagram

Application Code

Operating System

Redundant Hardware
Reliable Computing Platform

Digital Flight Control System

Control Application Domain

Continuous Differential Equations Model

Application Code

Reliable Computing Platform

Operating System

Replicated Processors

$P_1 \quad \ldots \quad P_r$

Sensors

$S_1 \quad \ldots \quad S_p$

Actuators

$A_1 \quad \ldots \quad A_q$
Research Objectives

- Establish hardware/software platform for ultra-reliable computing
- Use fault-tolerant computer architecture
- Use formal methods to prevent design and implementation errors
  - first specify in conventional mathematical notation
  - then specify and mechanically verify in EHDM
- Construct reliability model to quantify reliability estimate
Operating System for Control Applications

Single Processor Abstraction

Fault-tolerant Synchronous Replicated Model

Fault-tolerant Asynchronous Replicated Model

Hardware/Software Implementation
Application Task Characteristics

1. Fixed set of tasks
2. Hard deadlines
3. Multi-rate cyclic scheduling
4. Minimal jitter
5. Upper bound on task execution time
6. Precedence constraints
Architectural Concept

Sensors

Interactive Consistency Distribution Network

Processor Replicate 1

Interprocessor Communication Link

Processor Replicate R

Interprocessor Communication Link

...

Actuators
Design Decisions

1. the system is non-reconfigurable
2. the system is frame-synchronous
3. the scheduling is static, non-preemptive
4. internal voting is used which can recover the state of a processor affected by a transient fault within a bounded time
Reliability Modeling

Reliability model of quadruplex version of system

\[ \lambda_T = \text{transient fault rate (~ } 10^{-3}/\text{hr}) \]

\[ \lambda_p = \text{permanent fault rate (~ } 10^{-4}/\text{hr}) \]

\[ \rho = \text{rate of recovery from transient fault (design-dependent)} \]
Transient Fault Recovery

\[ P_f \]

Mean Time to Recover From Transient (hours)

Note inflection point on the order of one minute
$M$ frames = 1 cycle

$M_i > 0$ subframes per frame

$K$ tasks

$(i,j) = \text{cell (frame,subframe)}$

ST: scheduled task for cell $(i,j)$

TI: task inputs for cell $(i,j)$ \{tasks have no permanent state\}

AO: actuator output tasks

IR: initial task inputs
Uniprocessor Model

State of abstract machine given by:

\[
OS\_state = (\text{frame} : \{0..M - 1\}, \\
\quad \text{results} : \{0..M - 1\} \times \text{nat} \rightarrow D)
\]

The OS state transition is defined by the function \(OS\).

\[
OS : \text{Sin} \times OS\_state \rightarrow OS\_state
\]

\[
OS(s, u) = (u.\text{frame} \oplus 1, \lambda i, j. \text{new\_results}(s, u, i, j))
\]

where

\[
x \oplus y = (x + y) \mod M
\]

\[
x \ominus y = (x + M - y) \mod M
\]

\[
\text{new\_results}(s, u, i, j) = \begin{cases} 
\text{if } i = u.\text{frame} \\
\text{then } \text{exec}(s, u, i, j) \\
\text{else } u.\text{results}(i, j)
\end{cases}
\]
Uniprocessor Model (Cont’d.)

\[
\text{exec} : \text{Sin} \times \text{OS\_state} \times \{0..M - 1\} \times \text{nat} \to D
\]

\[
\text{exec}(s, u, i, j) = f_{ST(i,j)}(\text{arg}(T_1(i, j)[1], s, u, i, j), \ldots, \text{arg}(T_1(i, j)[n], s, u, i, j))
\]

\[
\text{arg} : \text{triple} \times \text{Sin} \times \text{OS\_state} \times \{0..M - 1\} \times \text{nat} \to D
\]

\[
\text{arg}(t, s, u, i, j) = \text{if } t.\text{type} = \text{sensor}
\]
\[
\text{then } s[t.i]
\]
\[
\text{else if } t.i = i \land t.j < j
\]
\[
\text{then } \text{exec}(s, u, i, t.j)
\]
\[
\text{else } u.\text{results}(t.i, t.j)
\]

Actuator output is a function of the OS state:

\[
UA(u) = \left[ \sum_{k=1}^{q} \text{Act}(u, k) \right]
\]

\[
\text{Act}(u, k) = \begin{cases} 
  u.\text{results}(\text{u\_frame } \Theta 1, j) & \text{if } \exists j : A_0(\text{u\_frame } \Theta 1, j) = k \\
  \phi & \text{otherwise}
\end{cases}
\]
Replicated Processor Model

The replicated processor model is based on a replicated state and transitions that allow for faults in the replicates

\[ \text{Repl} : ICin \times \text{Repl\_state} \times \text{fault\_status} \to \text{Repl\_state} \]

\[ \text{Repl}(c, r, \Phi) = \left[ \sum_{k=1}^{R} \text{RT}(c, r, k, \Phi) \right] \]

\[ \text{RT}(c, r, k, \Phi) = \text{if } \Phi[k] \text{ then } \bot \text{ else } (\text{frame\_vote}(r, \Phi), \text{Repl\_results}(c, r, k, \Phi)) \]

\[ \text{frame\_vote}(r, \Phi) = \text{maj}(\left[ \sum_{i=1}^{R} \text{FV}_i \right]) \]

where \( \text{FV}_i = \text{if } \Phi[i] \text{ then } \bot \text{ else } r[i].\text{frame} \oplus 1 \)

\[ \text{maj} : \text{sequence}(D \cup \{\bot\}) \to D \cup \{\bot\} \]
Replicated Processor Model (Cont’d.)

\[ VP : \{0..M - 1\} \times \text{nat} \times \{0..M - 1\} \rightarrow \{T, F\} \]

\[ VP(i, j, n) = T \text{ iff we are to vote } OS.results(i, j) \text{ during frame } n. \]

\[ Repl\_results(c, r, k, \Phi) = \]
\[ \lambda i, j. \text{ if } VP(i, j, r \cdot k) \cdot \text{frame} \]
\[ \text{ then results\_vote(c, r, i, j, } \Phi) \]
\[ \text{ else new\_results(c[k], r[k], i, j). } \]

\[ results\_vote(c, r, i, j, \Phi) = maj([_{l=1}^{R} RV_l]) \]

where \( RV_l = \text{ if } \Phi[l] \text{ then } \bot \text{ else new\_results(c[l], r[l], i, j). } \)

Replicated actuator output considers fault status indicators:

\[ RA : \text{Repl\_state} \times \text{fault\_status} \rightarrow RA_{out} \]

\[ RA(r, \Phi) = [_{k=1}^{R} RA_k] \]

where \( RA_k = \text{ if } \Phi[k] \text{ then } \bot \text{ else } UA(r[k]) \)
A Simple Fault Model

The results we seek must hold for all $\mathcal{F} : \{1..R\} \times \text{nat} \rightarrow \{T, F\}$ that satisfy a condition for maximally unfortunate fault behavior. Define a working processor as follows.

$$\mathcal{W} : \{1..R\} \times \text{nat} \times \text{fault_fn} \rightarrow \{T, F\}$$

$$\mathcal{W}(k, n, \mathcal{F}) = \forall j : 0 \leq j \leq \min(n, N_R) \supset \sim \mathcal{F}(k, n - j)$$

A processor that is nonfaulty, but not yet working, is considered to be recovering. The number of working processors is given by:

$$\omega(n, \mathcal{F}) = |\{k \mid \mathcal{W}(k, n, \mathcal{F})\}|$$

**Definition 1** The Maximum Fault Assumption for a given fault function $\mathcal{F}$ is that $\omega(n, \mathcal{F}) > R/2$ for every frame $n$.

All theorems about state machine correctness are predicated on this assumption.
Framework For Proving State Machine Correctness

Functions needed to bridge the gap between the two machines are those that do the following:

1. Map sensor inputs for $UM$ into replicated sensor inputs for $RM$.
2. Map replicated actuator outputs from $RM$ into actuator outputs for $UM$.
3. Map replicated OS states of $RM$ into uniprocessor OS states of $UM$.

Diagram: [Insert diagram here]
Correctness Criteria

**Definition 2** \( RM \) correctly implements \( UM \) under assumption \( P \) iff the following formula holds:

\[
\forall F : P(F) \supset \neg S, \forall n > 0 : a_n = \nu(b_n)
\]

(1)

where \( a_n \) and \( b_n \) can be characterized as functions of an initial state and all prior inputs.

We parameterize the concept of necessary assumptions using the predicate \( P \). For the replicated system, it will be instantiated by the Maximum Fault Assumption:

\[
P(F) = (\forall m : \omega(m, F) > R/2).
\]
Derived Correctness Criteria

**Definition 3 (Replicated OS Correctness Criteria)** \( RM \) correctly implements \( UM \) if the following conditions hold:

1. \( u_0 = \text{maj}(r_0) \)

2. \( \forall \mathcal{F}, (\forall m : \omega(m, \mathcal{F}) > R/2) \supset \forall S. \forall n > 0 : OS(s_n, \text{maj}(r_{n-1})) = \text{maj}(\text{Repl}(IC(s_n), r_{n-1}, \mathcal{F}_n^R)) \)

3. \( \forall \mathcal{F}, (\forall m : \omega(m, \mathcal{F}) > R/2) \supset \forall S. \forall n > 0 : UA(\text{maj}(r_n)) = \text{maj}(RA(r_n, \mathcal{F}_n^R)) \)
Sufficient Conditions for Correctness

Generic State Machine Correctness Criteria
→
Replicated OS Correctness Criteria
→
Consensus Property
→
Replicated State Invariant
→
Full Recovery Property
→
Voting Pattern
Intermediate Assertions

Definition 4 (Consensus Property) For $\mathcal{F}$ satisfying the Maximum Fault Assumption, the assertion

$$\forall p, n \in \mathbb{N}, n > 0. \quad \mathcal{W}(p, n - 1, \mathcal{F}) \supset r_{n-1}[p] = \text{maj}(r_{n-1}) \land r_{n}[p] = \text{maj}(r_{n})$$

holds for all $p$ and all $n > 0$.

Definition 5 (Replicated State Invariant) For fault function $\mathcal{F}$ satisfying the Maximum Fault Assumption, the following assertion is true for every frame $n$:

$$\forall p, n \in \mathbb{N}, n > 0. \quad (n = 0 \lor \sim \mathcal{F}(p, n - 1)) \supset$$

$$r_{n}[p].frame = \text{maj}(r_{n}).frame = n \mod M \land$$

$$\forall i, j : \text{rec}(i, j, \mathcal{L}(p, n, \mathcal{F}), \mathcal{H}(p, n, \mathcal{F}), T) \supset$$

$$r_{n}[p].results(i, j) = \text{maj}(r_{n}).results(i, j).$$
Recovery Concepts

Recovery of state element \((i, j)\) where last faulty frame was \(f\) and processor has been healthy for \(h\) frames:

\[
rec(i, j, f, h, e) = \begin{cases} 
F & \text{if } h \leq 1 \\
(VP(i, j, f \oplus h) \wedge e) \lor & \text{else}
\end{cases}
\]
\[
\text{if } i = f \oplus h
\]
\[
\text{then } \bigwedge_{l=1}^{TI(i,j)} RI(TI(i,j)[l], i, j, f, h)
\]
\[
\text{else } rec(i, j, f, h - 1, T)
\]

\[
RI(t, i, j, f, h) = (t.type = sensor) \lor 
\]
\[
\text{if } t.i = f \oplus h \land t.j < j
\]
\[
\text{then } rec(t.i, t.j, f, h, F)
\]
\[
\text{else } rec(t.i, t.j, f, h - 1, T)
\]

Definition 6 (Full Recovery Property) The predicate \(rec(i, j, f, N_R, T)\) holds for all \(i, j, f\).
Continuous Voting

\[ VP(i, j, k) = T \quad \forall i, j, k \]

\[ N_R = 2 \quad \text{(Actual } N_R = 1) \]

- Specifies that the *entire state* will be voted every frame
- Not very practical
- But proof is simple
Cyclic Voting

\[ VP(i, j, k) = (i = k) \quad \forall i, j, k \]

\[ N_R = M + 1. \]

- Only results just computed will be voted in a frame
- More practical
- Proof almost as simple

Frame

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>v</td>
<td></td>
<td></td>
<td></td>
<td>v</td>
<td></td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td></td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
</tbody>
</table>
Minimal Voting

- Vote only portion of state that will not be recovered from new sensor values.
- Construct $VP$ to ensure each cycle of graph is cut by at least one vote.
- $N_R = L_C + L_N + M$ where
  - $L_C$ = maximum frame length for all cycles
  - $L_N$ = maximum frame length for all noncyclic paths
Summary

- Ultra-reliable control systems hard to achieve
- Simple fault-tolerant design postulated
- Formal specification of design constructed
- Preliminary correctness proofs obtained
- Will extend from here
  - more sophisticated designs
  - mechanical verification
A Verified Model of Fault-Tolerance

John Rushby

Computer Science Laboratory
SRI International
Transient Faults are Common and Important

NASA-LaRC 1988 FCDS Technology Workshop:

- A number of DFCS are highly susceptible to radiated EM energy (composite materials provide less shielding, densely packed VLSI more susceptible to SEU)

- Designers must prove that their design will always recover from any and all non-hard faults reasonably quickly
Goal

- A model of a replicated system with exact-match voting

- A fault model that includes transients

- A theorem that establishes the conditions under which the system provides fault tolerance

- A formal specification of the model and a mechanically checked verification of the theorem that is consonant with the journal-level presentation
Status

- Model based closely on that developed by Butler, Caldwell, and DeVito at LaRC, but simplified and more abstract
  - Does not model frames and cycles
  - Does not model sensor failure or loss of frame counter
- Model and theorem described in draft journal-level report
- Specification and verification in Ehdm completed (Jim Caldwell provided stimulation and help in the proof)
- Currently reconciling the two
- Next step is to address the (over) simplifications
General Idea

\[ \text{sensors} \quad \text{actuator} \]

\[ \text{committed-to(c)} \]

\[ x = \text{members of foundation(c)} \]
Sets

sets: MODULE [T: TYPE]
EXPORTING ALL
THEORY

set: TYPE IS function[T -> bool]

x, y, z: VAR T
a, b, c: VAR set

union: function[set, set -> set] ==
    (LAMBDA a, b : (LAMBDA x : a(x) OR b(x)))

subset: function[set, set -> bool] =
    (LAMBDA a, b : (FORALL z : a(z) IMPLIES b(z)))

member: function[T, set -> bool] == (LAMBDA x, b : b(x))

empty: function[set -> bool] = (LAMBDA a : (FORALL x : NOT a(x)))

emptyset: set == (LAMBDA x : false)

fullset: set == (LAMBDA x : true)

extensionality: AXIOM
    (FORALL x : member(x, a) = member(x, b)) IMPLIES (a = b)
Cardinality

\[ m, n, p: \text{VAR nat} \]

\[ \text{card: function[set -> nat]} \]

\[ \text{card\_ax: AXIOM} \]
\[ \text{card(union(a, b)) + card(intersection(a, b)) = card(a) + card(b)} \]

\[ \text{card\_subset: AXIOM subset(a, b) IMPLIES card(a) \leq card(b)} \]

\[ \text{card\_empty: AXIOM card(a) = 0 IFF empty(a)} \]

\[ \text{empty\_prop: LEMMA card(a) > 0 IMPLIES (EXISTS x : member(x, a))} \]

\[ \text{card\_prop: LEMMA} \]
\[ \text{subset(a, c)} \]
\[ \text{AND subset(b, c)} \]
\[ \text{AND 2 * card(a) > card(c) AND 2 * card(b) > card(c)} \]
\[ \text{IMPLIES card(intersection(a, b)) > 0} \]
Sensors etc.

C: TYPE

a, c: VAR C

cell_types: TYPE = (sensor_cell, actuator_cell, task_cell).
cell_type: function[C -> cell_types]
sensors: TYPE FROM C WITH (LAMBDA c : cell_type(c) = sensor_cell)
actuators: TYPE FROM C WITH (LAMBDA c : cell_type(c) = actuator_cell)
active_tasks: TYPE FROM C WITH
  (LAMBDA c : cell_type(c) /= sensor_cell)
voted: TYPE FROM C

voted_ax: AXIOM
  (c IN actuators IMPLIES c IN voted)
  AND (c IN voted IMPLIES NOT (c IN sensors))

Gbar: function[C, C -> bool]

sensor_ax: AXIOM (EXISTS a : Gbar(a, c)) IFF NOT (c IN sensors)
Simple Machine

\[ step(\sigma, c, n) = \sigma \text{ with } [c := \text{if } c \in C \text{-} S \text{ then } \text{sensor}(c)(n) \text{ else } \text{task}(c)(\sigma)] \]

\[ \begin{align*}
run(0) &= (\lambda c : \bot) \\
run(n + 1) &= step(run(n), \text{sched}(n + 1), n + 1).
\end{align*} \]

\textbf{step: function}[\text{state, C, M} \rightarrow \text{state}] = \\
(\text{LAMBDA } s, c, m : s \\
\text{WITH } [c := \\
\text{IF } c \text{ IN sensors THEN } \text{sensor}(c)(m) \text{ ELSE } \text{task}(c)(s) \text{ END IF}])

\textbf{identity: function}[\text{M} \rightarrow \text{nat}] = (\text{LAMBDA } m : m)

\textbf{run: RECURSIVE function}[\text{M} \rightarrow \text{state}] = \\
(\text{LAMBDA } m : \\
\text{IF } m = 0 \text{ THEN undef ELSE } step(\text{run}(m - 1), \text{sched}(m), m) \text{ END IF})
\text{BY identity}
TCC's

(* Subtype TCC generated for the first argument to task in dependency *)

dependency_TCC1: FORMULA
(c IN active_tasks AND (FORALL a : Gbar(a, c) IMPLIES s(a) = t(a)))
IMPLIES (cell_type(c) /= sensor_cell)

(* Subtype TCC generated for the first argument to sensor in step *)

step_TCC1: FORMULA (c IN sensors) IMPLIES (cell_type(c) = sensor_cell)

(* Subtype TCC generated for the first argument to task in step *)

step_TCC2: FORMULA
(NOT (c IN sensors)) IMPLIES (cell_type(c) /= sensor_cell)

(* Subtype TCC generated for the first argument to run in run *)

run_TCC1: FORMULA (m >= 0) IMPLIES (NOT (m = 0)) IMPLIES (m - 1 >= 0)

(* Termination TCC generated for run *)

run_TCC2: FORMULA
(m >= 0) IMPLIES (NOT (m = 0)) IMPLIES identity(m) > identity(m - 1)
Replicated Machine

\[ \neg\mathcal{F}(i)(n) \supset \text{sstep}(\rho, c, n)(i) = \text{step}(\rho(i), c, n), \]

\[ \neg\mathcal{F}(i)(n) \supset \text{vote}(\rho, c, n)(i) = \text{if } c \in C_V \text{ then } \rho(i) \text{ with } [c := \text{maj}\{\rho(j)(c) | j \in R}\} \]

\text{else } \rho(i) \]

\[ r\text{step}(\rho, c, n) = \text{vote}(\text{sstep}(\rho, c, n), c, n). \]
Replicated Machine

sstep_ax: AXIOM
    NOT (F(i)(m)) IMPLIES sstep(rs, c, m)(i) = step(rs(i), c, m)

maj_ax: AXIOM
    (EXISTS A :
        2 * card(A) > card(fullset[R])
        AND (FORALL i : member(i, A) IMPLIES rs(i)(c) = x))
    IMPLIES maj(rs, c) = x

vote_ax: AXIOM
    NOT (F(i)(m))
    IMPLIES vote(rs, c, m)
       = IF c IN voted
          THEN rs WITH [(i)(c) := maj(rs, c)]
       ELSE rs END IF

rstep: function[rstate, C, M -> rstate] ==
    (LAMBDA rs, c, m : vote(sstep(rs, c, m), c, m))

rrun: RECURSIVE function[M -> rstate] =
    (LAMBDA m :
        IF m = 0
        THEN (LAMBDA i : undef)
        ELSE rstep(rrun(m - 1), sched(m), m) END IF)
    BY identity
Foundation etc.

\[
\text{foundation}(c) = \begin{cases} 
\{c\} & \text{if } c \in (C_S \cup C_V) \\
\{c\} \cup \bigcup_{(a,c) \in \overline{G}} \text{foundation}(a) & \text{otherwise}
\end{cases}
\]

\[
\text{support}(c) = \begin{cases} 
\{c\} \cup \bigcup_{(a,c) \in \overline{G}} \text{foundation}(a) & \text{if } c \in C_V \\
\text{foundation}(c) & \text{otherwise.}
\end{cases}
\]

\[
\text{committed-to}(c) = \min\{\text{when}(a) | a \in \text{support}(c)\}.
\]
Foundation etc.

foundation: RECURSIVE function[C -> set[C]] =
    (LAMBDA c :
        (LAMBDA a : c = a
            OR (NOT (c IN voted OR c IN sensors)
            AND (EXISTS b :
                Gbar(b, c) AND member(a, foundation(b))))))
    BY dowhen

backup: function[C -> set[C]] =
    (LAMBDA c :
        (LAMBDA a :
            (EXISTS b : Gbar(b, c) AND member(a, foundation(b)))))

support: function[C -> set[C]] =
    (LAMBDA c :
        (LAMBDA a :
            member(a, foundation(c))
            OR (c IN voted AND member(a, backup(c)))))

critical_times: function[C -> set[M]] ==
    (LAMBDA c : (LAMBDA t : member(sched(t), support(c))))

committed_to: function[C -> M] == (LAMBDA c : min(critical_times(c)))
$OK(i)(c) = (\forall n : \text{committed-to}(c) \leq n \leq \text{when}(c) \supset \neg \mathcal{F}(i)(n))$.

$MOK(c) = \exists \Theta \subseteq R, |\Theta| > r/2, i \in \Theta \supset OK(i)(c)$.

**OK**: function[R -> set[C]] =
(LAMBDA i ::
  (LAMBDA c ::
   (FORALL m ::
     committed_to(c) <= m AND m <= dowhen(c)
     IMPLIES NOT F(i)(m))))

**working**: function[C -> set[R]] = (LAMBDA c :: (LAMBDA i :: OK(i)(c)))

**MOK**: function[C -> bool] =
(LAMBDA c :: 2 * card(working(c)) > card(fullset[R]))
Theorem

If

\[ \forall a : \text{when}(a) \leq \text{when}(c) \supset MOK(a), \]

then

\[ \forall j : OK(j)(c) \supset rrunto(c)(j)(c) = runto(c)(c). \]

safe: RECURSIVE function[C \to bool] =
  (LAMBDA c : MOK(c) AND (FORALL a : Gbar(a, c) IMPLIES safe(a)))
  BY dowhen

correct: function[C \to bool] =
  (LAMBDA c :
    (FORALL j : OK(j)(c) IMPLIES rrunto(c)(j)(c) = runto(c)(c)))

the_result: THEOREM safe(c) IMPLIES correct(c)
Noetherian Induction

noetherian: MODULE [dom: TYPE, <: function[dom, dom -> bool]]
ASSUMING
  measure: VAR function[dom -> nat]
a, b: VAR dom

well_founded: FORMULA
  (EXISTS measure : a < b IMPLIES measure(a) < measure(b))
THEORY
  p, A, B: VAR function[dom -> bool]
d, d1, d2, d3, d4: VAR dom

general_induction: AXIOM
  (FORALL d1 : (FORALL d2 : d2 < d1 IMPLIES p(d2)) IMPLIES p(d1))
  IMPLIES (FORALL d : p(d))

mod_induction: THEOREM
  (FORALL d3, d4 : d4 < d3 IMPLIES A(d3) IMPLIES A(d4))
  AND (FORALL d1 :
    (FORALL d2 : d2 < d1 IMPLIES (A(d1) AND B(d2)))
    IMPLIES B(d1))
  IMPLIES (FORALL d : A(d) IMPLIES B(d))
PROOF
  mod_proof: PROVE mod_induction d1 <- d1@p1, d3 <- d1@p1, d4 <- d2
  FROM general_induction p <- (LAMBDA d : A(d) IMPLIES B(d))
END noetherian
The Proof

correctness_proof: MODULE
USING correctness, voted_step, nonvoted_step, sensor_step,
   noetherian[C, Gbar]
PROOF
   a, c: VAR C

   discharge_well_founded: PROVE well_founded measure <- dowhen FROM
      Gbar_when c <- b

   inductive_step: LEMMA
      (FORALL a : Gbar(a, c) IMPLIES safe(c) AND correct(a))
      IMPLIES correct(c)

   almost_final_proof: PROVE inductive_step a <- a@p7 FROM
      sensor_inductive_step, voted_inductive_step, nonvoted_inductive_step,
      induction_body a <- a@p1, induction_body a <- a@p2,
      induction_body a <- a@p3, induction_body

   final_proof: PROVE the_result FROM
      mod_induction A <- safe, B <- correct, d <- c, d2 <- a@p3,
      safe a <- d4@p1, c <- d3@p1,
      inductive_step c <- d1@p1

END correctness_proof
Summary

- Formal specification and verification revealed typos in the original report

- Exposed omission in original proof

- Led to stronger theorem and more elegant proof (using Noetherian rather than ordinary induction)

- Confirmed that Ehdm has the capability to specify interesting and useful properties in a direct, natural, and readable manner

- Proofs were hard (three intensive man-weeks, 92 lemmas); I haven't yet gone back to see why that was so

- We have the beginnings of a formally verified model for a fault tolerant operating system
The Design and Proof of Correctness of a Fault-Tolerant Circuit

William R. Bevier
William D. Young

Computational Logic, Inc.
1717 W. 6th Street
Austin, Texas

18 August 1990
What We Accomplished

• A formal statement of Interactive Consistency Conditions\(^1\) in the Boyer-Moore logic.

• A formal statement of the Oral Messages algorithm \(OM\) in the Boyer-Moore logic.

• A mechanically checked proof that \(OM\) satisfies the Interactive Consistency conditions.

• A mechanically checked proof of the optimality result: no algorithm can tolerate fewer faults than \(OM\) yet still achieve Interactive Consistency.

• The use of \(OM\) in a functional specification for a fault-tolerant device.

• A formal description of the design of the device.

• A mechanically checked proof that the device design satisfies the specification.

• An implementation of the design in programmable logic arrays.

A Stack of Related Machines

spec

design

implementation
The Specification

The specification is a function that describes a finite state machine.

At every step, each of $N$ processes

1. reads its sensor input,

2. exchanges its sensor value with all other processes,

3. produces an interactive consistency vector (ICV) that contains what it concludes is each other process's value, and

4. applies a filter function to the ICV to produce an output.
Properties of the Specification Function

The exchange of sensor values is accomplished by an algorithm called $OM$.

$OM$ achieves *interactive consistency*. That is,

A process sends a message to $n-1$ destination processes.
1. All non-faulty destination processes agree on the same received value.
2. If the sending process is non-faulty, then every non-faulty destination process receives the message sent.

$OM$ has been defined as a function in the Boyer-Moore logic, and a proof that interactive consistency is achieved has been mechanically checked.
Formal Statement of Correctness of $OM$

Let

- $n$ be the number of processes,
- $L$ be the set $\{0, ..., n-1\}$,
- $g, i, j \in L$ be process names,
- $x$ be $g$'s local value, and
- $m$ give the number of rounds of information exchange.

The interactive consistency conditions are stated as follows.

\[ \neg faulty(i) \]
\[ \& \neg faulty(j) \]
\[ \& 3 \text{ faults}(L) < n \]
\[ \& \text{ faults}(L) \leq m \]
\[ \rightarrow \]
\[ OM(n, g, x, m)[i] = OM(n, g, x, m)[j]. \]

\[ \neg faulty(g) \]
\[ \& \neg faulty(i) \]
\[ \& 3 \text{ faults}(L) < n \]
\[ \& \text{ faults}(L) \leq m \]
\[ \rightarrow \]
\[ OM(n, g, x, m)[i] = x \]
Specification Abstraction

The following aspects of the specification are not constrained:

1. The number of processes.

2. The types of the input and output values.

3. The nature of the filter function.
What Interactive Consistency Guarantees

The specification can be thought of as a function which
- receives a sequence of $N$-tuples of input values, and
- produces a sequence of $N$-tuples of output values.

Because of Interactive Consistency, we can conclude:

At each step, all non-faulty processes agree on their output iff the total number of processors exceeds three times the number of faulty processors.
The Device Design

Goal: Design 4 identical circuits which, when operating synchronously, achieve Byzantine agreement.
A Process Internal State
Process Steps

0: \text{data\_out}[i] \leftarrow \text{sense}, \; i \in \{0, 1, 2\}
\text{icv}[3] \leftarrow \text{sense}
\text{clock} \leftarrow \text{clock}+1

1: \text{m}[0, i] \leftarrow \text{input}[i], \; i \in \{0, 1, 2\}
\text{data\_out}[0] \leftarrow \text{input}[1]
\text{data\_out}[1] \leftarrow \text{input}[0]
\text{data\_out}[2] \leftarrow \text{input}[0]
\text{clock} \leftarrow \text{clock}+1

2: \text{m}[1, i] \leftarrow \text{input}[i], \; i \in \{0, 1, 2\}
\text{data\_out}[0] \leftarrow \text{m}[0, 2]
\text{data\_out}[1] \leftarrow \text{m}[0, 2]
\text{data\_out}[2] \leftarrow \text{m}[0, 1]
\text{clock} \leftarrow \text{clock}+1

3: \text{m}[2, i] \leftarrow \text{input}[i], \; i \in \{0, 1, 2\}
\text{clock} \leftarrow \text{clock}+1

4: \text{icv}[0] \leftarrow \text{majority}(\text{m}[0, 0], \; \text{m}[1, 2], \; \text{m}[2, 1])
\text{icv}[1] \leftarrow \text{majority}(\text{m}[0, 1], \; \text{m}[1, 0], \; \text{m}[2, 2])
\text{icv}[2] \leftarrow \text{majority}(\text{m}[0, 2], \; \text{m}[1, 1], \; \text{m}[2, 0])
\text{clock} \leftarrow \text{clock}+1

5: \text{Actuator}
\text{clock} \leftarrow \text{filter}(\text{icv})
\text{clock} \leftarrow \text{clock}+1

6: \text{clock} \leftarrow \text{clock}+1

7: \text{clock} \leftarrow \text{clock}+1
Summary of Device Design

1. Four identical devices.

2. Only internal and external data flow specified, data width not.

3. Filter function constrained to tolerate ICV rotations.
Correctness of Device Design
Device Implementation

by Larry Smith
Verifying an Interactive Consistency Circuit:
A Case Study in the Reuse of a Verification Technology

Mark Bickford
Mandayam Srivas

Odyssey Research Associates, Inc.
301A Harris B. Dates Drive
Ithaca, NY 14850.

This talk presented the work done at ORA for NASA-LRC in the design and formal verification of a hardware implementation of a scheme for attaining interactive consistency (byzantine agreement) among four microprocessors. The microprocessors used in the design are an updated version of a formally verified 32-bit, instruction-pipelined, RISC processor, MiniCayuga. The 4-processor system, which is designed under the assumption that the clocks of all the processors are synchronized, provides "software control" over the interactive consistency operation. Interactive consistency computation is supported as an explicit instruction on each of the microprocessors. An identical user program executing on each of the processors decides when and on what data interactive consistency must be performed.

This exercise also served as a case study to investigate the effectiveness of reusing the technology which had been developed during the MiniCayuga effort for verifying synchronous hardware designs. MiniCayuga was verified using the verification system Clio which was also developed at ORA. To assist in reusing this technology a computer-aided specification and verification tool was developed. This tool specializes Clio to synchronous hardware designs and significantly reduces the tedium involved in verifying such designs. The talk presented the tool and described how it was used to specify and verify the interactive consistency circuit.
Summary

Achievements

1. Formalization of abstract Byzantine agreement algorithm.

2. Use of this algorithm to specify a hardware device.

3. A mechanically checked proof that the device design is correct.

4. The implementation of the device form the low-level design.

Limitations

1. Assumes synchronized behavior of the processes.
Verifying an Interactive Consistency Circuit:  
A Case Study in the Reuse of a Verification Technology

Mark Bickford
Mandayam Srivas

Odyssey Research Associates, Inc.
301A Harris B. Dates Drive
Ithaca, NY 14850.
Objectives of the Work

- Design an efficient hardware implementation for a 4-processor architecture
- Use verified MiniCayuga’s in the design
- Verify the design
- Reuse MiniCayuga verification technology
  - A method of modeling synchronous hardware designs in the Clio verification system
  - Formalizing a class of properties most commonly encountered in verifying designs
  - A "standard" proof strategy
Clio: A functional Language Based Verification System

- Caliban: A modern functional language
  e.g., higher order functions, data types, lazy, etc.

  \[
  \text{least } P \ x = x, \ P \ x \\
  \text{least } P \ x+1
  \]

- Assertion Level: Full FOPL with equality
  on Caliban terms

  \[
  \text{Prop} := (\exists) (x) \land \left[ (\forall)(\text{least } P \ x) = \text{`True`} \right] \\
  \lor \ `P (\text{least } P \ x)` = `True`
  \]

- Interactive Theorem Prover
  - rewriting
  - Induction
    - structural
    - Fixed point
  - Other FOPL proof strategies
Presentation Outline

- IC circuit design

- The computer-aided hardware verification tool

- How we verified it

- General observations about the effort
The Hardware Design: Overview
Two new instructions:

ICOP REG - initiates and co-ordinates IC computation
MOVE SREG REG - moves special REG to general REG

|| check if voter is free
Notfree MOVE STATUS REG1
JIF REG1 Notfree
ICOP REG2

|| check if IC computation is complete
Notready MOVE STATUS REG1
JIF REG1 Notready

|| move the results of IC to general registers
MOVE SREG0 REG3
MOVE SREG1 REG4
MOVE SREG2 REG5
The Hardware Design: Overview
• voter separate from processor: modularity

• point-to-point connection: electrical isolation

• serialize data transfers: number of pins Vs. time

• Fault region: processor, voter, and the connections they feed
• no absolute indexing scheme for processors/voters
  – relative indexing scheme: \( \text{succ}, \text{succ}^2, \text{succ}^3 \)

  – IC vectors will be stored in the processors in the order of their successors

• Underlying assumption: clocks are synchronized with at most a bounded skew

  – hold sender’s signal stable for one phase longer than needed
IC System Design Behavior

- **Initiate**: draw the attention of voter (1)
- **Load**: transfer private values (2)
- **Exchange**: exchange received values (6)
- **Compute**: compute and store IC vector (3)
MiniCayuga Processor: Summary

- Inspired by Cayuga (Cornell University)

- 32-bit RISC processor

- Design characteristics
  - 32 general purpose registers
  - small and simple instruction set
  - 3-stage instruction pipeline: fetch, compute, writeback
  - delayed jump, pipeline stalling, internal forwarding
  - interrupt
What do we prove?

Assuming

- every Cayuga-FT is about to execute an ICOP,
- every Voter is ready to vote, and
- there is at most one faulty region,

then, 12 cycles later the system state will satisfy the following conditions:

- The IC vectors in the processors are identical "up to rotation."
- The IC vectors are correct w.r.t. to the processor private values 12 cycles earlier.
A Computer-Aided Verification Tool

- Specializes Clio to the domain of finite state controller systems
- Design specification generation
- Verification condition formulation
- Automatic proof support
The Voter Circuit

Controller State Machine
Finite State Controller Systems (FSCS)

- Central Controller + Data Path components

- Component behavior is specified as a set of actions

- Controller is specified as an FSM which schedules a set of actions on the components.

- Timing Model
  - Every transition corresponds to a clock cycle (with multiple phases)
  
  - An action may have zero or more units (phases) of delay
  
  - Actions are synchronized with state transitions
Specification technology reused

- a method of formalizing the intended operational model of an FSCS in Caliban/Clio

designspecgen ::
    data-path-structure ->
    controller-structure ->
    controller-schedule ->
    actions-behavior -> design-spec

Execute :: STATE -> STATE

"single clock cycle behavior of design"
Proof technology shared

- Form of the most commonly proved conditions
  - Invariant conditions

- Advance conditions

- Proof strategy: "controlled symbolic evaluation (rewriting) with selective case-splits"
The Specification Hierarchy
Rationale for the hierarchy

- Decompose proofs into manageable units

- Need for the black level
  - introduce "error" actions
  - type of Execute is different from that of action

- Implication of intermediate levels
  - pro: proof can take "bigger" steps
  - con: must come up with intermediate abstract specification
Top Level Specification

||IcNetState ::= <<(INDEX -> FTCstate),
  (INDEX -> Voterstate), Interrupts>>

IcNetStep <<ftc,vtr, int:rest>> =
  <<newftc,newvtr ,rest>>
  where newftc index
    = fault_ftc_step index ftc (ftcinput index)
    newvtr index
    = fault_vtr_step index vtr (vtrinput index)
    ftcinput index
    = make_ftc_in (select_int index int)
      (fault_to_proc index ftc vtr)
    vtrinput index
    = Voterinput index ftc vtr
      (ftcinput index )

fault_ftc_step index s in =
  FtCayugaStep (s index) in , ~(faulty index)
  byzCayugaStep (s index) in

fault_vtr_step index s =
  voterstep (s index) , ~(faulty index)
  byzstep (s index)

...
Formal Statement of Correctness

MainTheorem :=
    Preconditions 's' => ResultConsistent 's'

ResultConsistent 's' :=
    Consistent 'icvec s (Iterate #12 IcNetStep s)'

Consistent 'array' :=
    'faulty index'=False' =>
        IndexConsistent 'array' 'index'

IndexConsistent 'array' 'index' :=

    ('faulty (succ index)'='False'=>
        '(array index).succ'='array (succ index)')

    & ('faulty (succ2 index)'='False'=>
        '(array index).succ2'='array (succ2 index)')

    & ('faulty (succ3 index)'='False'=>
        '(array index).succ3'='array (succ3 index)')
Preconditions 's' :=

  Proper_icnet 's' & Sync 'LDP1' 's' & All_go 's'

Sync 'cs' '<<ftc,vtr,inlist>>' :=
  ('faulty ONE' = 'False' =>
   'control (vtr ONE)'='cs')
& ('faulty TWO' = 'False' =>
   'control (vtr TWO)'='cs')
& ('faulty THREE' = 'False' =>
   'control (vtr THREE)'='cs')
& ('faulty FOUR' = 'False' =>
   'control (vtr FOUR)'='cs')

All_go 's' :=
  ('faulty ONE'='False' =>
   ('go_of (vtr s ONE)'='False' & 'go_signal s ONE'='False')
& ('faulty TWO'='False' =>
   ('go_of (vtr s TWO)'='False' & 'go_signal s TWO'='False')
& ('faulty THREE'='False' =>
   ('go_of (vtr s THREE)'='False' & 'go_signal s THREE'='False')
& ('faulty FOUR'='False' =>
   ('go_of (vtr s FOUR)'='False' & 'go_signal s FOUR'='False'))
Preconditions 's' :=
Proper_icnet 's' & Sync 'LDP1' 's' & All_go 's'

Sync 'cs' '<<ftc,vtr,inlist>>' :=
('faulty ONE' = 'False' =>
'control (vtr ONE)'='cs')
& ('faulty TWO' = 'False' =>
'control (vtr TWO)'='cs')
& ('faulty THREE' = 'False' =>
'control (vtr THREE)'='cs')
& ('faulty FOUR' = 'False' =>
'control (vtr FOUR)'='cs')

All_go 's' :=
('faulty ONE'='False' =>
('go_of (vtr s ONE)'='False' & 'go_signal s ONE'='GO'))
& ('faulty TWO'='False' =>
('go_of (vtr s TWO)'='False' & 'go_signal s TWO'='GO'))
& ('faulty THREE'='False' =>
('go_of (vtr s THREE)'='False'
& 'go_signal s THREE'='GO'))
& ('faulty FOUR'='False' =>
('go_of (vtr s FOUR)'='False'
& 'go_signal s FOUR'='GO'))
The proof strategy reused
"controlled symbolic execution of design"

1. Instantiate the states of components and inputs with appropriate symbolic constants.

2. Add all the conditions on the constants implied by the preconditions of the theorem as hypothesis.


4. Try *case-splitting* on all the conditionals automatically.

5. If either of the previous two steps seem to take too long, then case-split on the controller states and inputs before symbolic evaluation (step 3).
New technology needed

- Modeling faulty behavior

- Specification
  - determining the right hierarchy
  - writing intermediate "abstract" spec
  - defining abstraction function (ABS)

- Proof: "design level properness" implies "abstract level properness"
General Observations

- An engineering-oriented verification experience
  Lilith → MiniCayuga → IC circuit

- Methodology: top-down + bottom-up

- Level of effort: 1 man year

  - building the tool
  - developing designs
  - verification
Verification Effort Milestones

- formulated a top level correctness statement

- designed and verified a simple voter circuit

- specified voter and processor for a continuous voting scheme

- designed and verified second voter design
• discovered continuous voting scheme was "hard to synchronize"

• respecified voter and processor for a voting-on-demand scheme

• redesign and reverify voter

• verified overall system

• verified processor
• To integrate theorem proving based verification technology into the design process we need:
  
  – more machine assistance
  – domain specialization

• The next step?
  
  – A useful way of reporting failed proof attempts
  – Interaction with motivated and patient engineering design teams and projects
Hardware Verification at Computational Logic, Inc.

Bishop C. Brock
Warren A. Hunt, Jr.

Computational Logic Incorporated
1717 West Sixth Street, Suite 290
Austin, Texas 78703-4776

+1 512 322 9951
Brock@CLI.COM, Hunt@CLI.COM

3 August 1990
Talk Topics

- Hardware Verification: What Is It?
- Formal Methods: What Good Are They?
- Verification Methodology
- Present Accomplishments
- Expected Near Term Results
- Present Trends
- Future Directions
- Collaborations and Technology Transfer
- Technology Enablers
- Conclusions
Hardware Verification: What Is It?

The mathematical formalization of the specification of any (all) aspects of hardware design.

We specifically are interested in the design of hardware for digital computing.

Goals:

- Completely replace programmer's manuals, timing diagrams, interface specifications, power requirements, etc. with clear precise formulas.

- Provide a perfectly clear foundation upon which systems can be built.
Formal Methods: What Good Are They?

Formal methods in the U.S. have a bad credit rating.

Over the years, good mechanized software verification systems have been constructed.

Good software verification tools are being extended to include hardware verification, thus providing good systems verification tools.

Hardware verification seems more tractable than software verification:

- few, repeatedly-used, low-level constructs;
- specification domain is less abstract (fairly concrete); and
- formal methods can be used incrementally.

Last point is critical, note Bryant's work.
Our Verification Methodology

We employ the Boyer-Moore logic to:

• write design specifications;
• write behavioral specifications; and
• record relations.

The Boyer-Moore theorem prover
• insures that definitions are well formed;
• checks that proofs are correct; and
• manages our evolving database of facts.
Present Accomplishments

Our application of formal methods to hardware specification and verification include:

• Core RISC specification;
• FM8502 microprocessor verification;
• verification of circuits using standard TTL components;
• a formalization of a simple HDL; and
• verified synthesis of combinational circuits.

Let us consider several in more detail.
Core RISC

Bill Bevier has formally specified a set of instructions that characterize a Core RISC-compliant processor. This formalization includes:

- byte, half-word, and long-word memory accesses;
- Boolean, natural number, and integer ALU operations;
- a minimum register set; and
- an exception mechanism.

The emphasis here has been on mathematically modeling the instruction set.

Our study of RISC architectures indicates that we need to be able to model multi-phase clocking schemes before we attempt to design a build a verified Core RISC processor. This effort is ongoing.
The FM8502 Fabrication

Currently, our primary effort involves the fabrication of the FM8502 microprocessor.

This fabrication effort is a test-of-concept; that is, can we manufacture formally modeled circuits and get them working?

The FM8502 microprocessor is a 32-bit general purpose microprocessor with:

• 32-bit addressing;
• 16 general-purpose registers;
• two-address architecture;
• 5 addressing modes;
• a 16-function ALU
• extensive flag support; and
• little else.
### MODE OPERAND DESCRIPTION

<table>
<thead>
<tr>
<th>Mode</th>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Rn</td>
<td>Register Direct</td>
</tr>
<tr>
<td>01</td>
<td>(Rn)</td>
<td>Register Indirect</td>
</tr>
<tr>
<td>10</td>
<td>-(Rn)</td>
<td>Register Indirect Pre-decrement</td>
</tr>
<tr>
<td>11</td>
<td>(Rn)+</td>
<td>Register Indirect Post-increment</td>
</tr>
</tbody>
</table>

### OP-CODE OPERATION DESCRIPTION STORE-CC CONDITION

<table>
<thead>
<tr>
<th>OP-CODE</th>
<th>OPERATION</th>
<th>DESCRIPTION</th>
<th>STORE-CC</th>
<th>CONDITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>b &lt; a</td>
<td>Move</td>
<td>0000</td>
<td>Carry clear</td>
</tr>
<tr>
<td>0001</td>
<td>b &lt; a + 1</td>
<td>Increment</td>
<td>0001</td>
<td>Carry set</td>
</tr>
<tr>
<td>0010</td>
<td>b &lt; a + b + c</td>
<td>Add with carry</td>
<td>0010</td>
<td>Overflow clear</td>
</tr>
<tr>
<td>0011</td>
<td>b &lt; b + a</td>
<td>Add</td>
<td>0011</td>
<td>Overflow set</td>
</tr>
<tr>
<td>0100</td>
<td>b &lt; 0 - a</td>
<td>Negation</td>
<td>0101</td>
<td>Not negative</td>
</tr>
<tr>
<td>0101</td>
<td>b &lt; a - 1</td>
<td>Decrement</td>
<td>0101</td>
<td>Negative</td>
</tr>
<tr>
<td>0110</td>
<td>b &lt; a - b - c</td>
<td>Subtract with borrow</td>
<td>0110</td>
<td>Not zero</td>
</tr>
<tr>
<td>0111</td>
<td>b &lt; b - a</td>
<td>Subtract</td>
<td>0111</td>
<td>Zero</td>
</tr>
<tr>
<td>1000</td>
<td>b &lt; a &gt;&gt; 1</td>
<td>Rotate right through carry</td>
<td>1000</td>
<td>Higher</td>
</tr>
<tr>
<td>1001</td>
<td>b &lt; a &gt;&gt; 1</td>
<td>Arithmetic shift right</td>
<td>1001</td>
<td>Lower or same</td>
</tr>
<tr>
<td>1010</td>
<td>b &lt; a &gt;&gt; 1</td>
<td>Logical shift right</td>
<td>1010</td>
<td>Greater or equal</td>
</tr>
<tr>
<td>1011</td>
<td>b &lt; b XOR a</td>
<td>XOR</td>
<td>1011</td>
<td>Less</td>
</tr>
<tr>
<td>1100</td>
<td>b &lt; b OR a</td>
<td>OR</td>
<td>1100</td>
<td>Greater</td>
</tr>
<tr>
<td>1101</td>
<td>b &lt; b AND a</td>
<td>AND</td>
<td>1101</td>
<td>Less or equal</td>
</tr>
<tr>
<td>1110</td>
<td>b &lt; NOT a</td>
<td>NOT</td>
<td>1110</td>
<td>True</td>
</tr>
<tr>
<td>1111</td>
<td>b &lt; a</td>
<td>Move</td>
<td>1111</td>
<td>False</td>
</tr>
</tbody>
</table>
The FM8502 Implementation Specification

To be able to manufacture the FM8502 with some precision, we have been working on the formalization of an HDL.

We will prove the correctness of our HDL description of the FM8502, and then translate our HDL description into a commercial HDL.

Our HDL provides our lowest-level model for the FM8502 implementation:

- every internal gate and register is described;
- every I/O pad is defined; and
- we expect to validate our test vectors directly on our HDL description.

Our HDL specification also includes all of the internal test logic.
The FM8502 Pinout

Below is a pictorial diagram of the FM8502 pinout. Quite a number of pins are allocated to testing purposes.
A Formal HDL

Our HDL is structured like commercial HDL's:

- netlist based;
- heirarchicaly structured;
- occurrence-oriented; and
- allows multiple views of circuits.

We have a formal specification of our HDL:

- a predicate recognizes well-formed circuits; and
- several interpreters define the semantics.
HDL Examples of Circuits

'(HALF-ADDER (A B))
(SUM CARRY)
(( G0(SUM) B-XOR(A B))
 ( G1(CARRY) B-AND(A B)))

The following full-adder specification refers twice to the half-adder specification above.

'(FULL-ADDER (A B C))
(SUM CARRY)
(( T0(SUM1 CARRY1) HALF-ADDER(A B))
 ( T1(SUM CARRY2) HALF-ADDER(SUM1 C))
 ( T2(CARRY) B-OR(CARRY1 CARRY2)))
Verified Synthesis

We perform synthesis by

- writing circuit generator programs;
- verifying the circuit generator programs; and
- then running the generators to produce provably correct circuits.

In other words, after a circuit has been generated we need not inspect it for the Boolean correctness.
An ALU Generator

We have an arbitrary size, 16-function ALU generator which is:

- programmable -- ALUs with different internal structure can be produced;
- "intelligent" -- internal buffers are only added when needed; and
- has been verified to generate correct $n$-bit, gate-level ALU descriptions.

Simple translators can convert the ALU descriptions into conventional CAD languages (e.g., VHDL).

To replay the proof only takes about 20 (Sun 3) minutes.
ALU Generator Output Summary

Summarized below are some characteristics of the ALUs generated by our verified ALU generator.

<table>
<thead>
<tr>
<th>Size</th>
<th>Gate Count</th>
<th>Fanout</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>126</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>2 bits</td>
<td>149</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>4 bits</td>
<td>196</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>8 bits</td>
<td>297</td>
<td>8</td>
<td>22</td>
</tr>
<tr>
<td>16 bits</td>
<td>491</td>
<td>8</td>
<td>26</td>
</tr>
<tr>
<td>32 bits</td>
<td>880</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>64 bits</td>
<td>1665</td>
<td>8</td>
<td>35</td>
</tr>
<tr>
<td>128 bits</td>
<td>3227</td>
<td>8</td>
<td>39</td>
</tr>
</tbody>
</table>

Payoff: It only takes 0.6 seconds to generate a correct 32-bit ALU, 1.3 seconds for a 64-bit ALU, and 3.1 seconds for a 128-bit ALU.
Expected Near Term Results

Several projects underway which will conclude this year are:

- an ability to verify sequential circuits generators; and
- the fabrication of the FM8502 microprocessor.

We are using both combinational and sequential logic synthesis techniques in the fabrication of the FM8502.

We will be able to generate a correct $n$-bit microprocessor (so long as the word size is large enough to contain FM8502 instructions.)

We will generate a gate-array specification directly.

We are generating our test-vectors directly from our formal circuit specifications.

3 August 1990
Present Trends

There is increasing interest in:

• boolean comparison -- which should lead the way to more general purpose techniques;

• register-transfer specifications with circuit verification;

• formalization of self-timed circuits;

• formalization of timing behavior; and

• transformational systems.

These trends are all indicative of increased use of formal techniques for hardware specification and verification.

And these techniques are being applied incrementally.
Future Directions

In the future we hope to:

- formalize a subset of VHDL (using our Ada formalization experience);
- perform tool verification (e.g., logic minimizer, tautology checkers);
- verify a Core RISC microprocessor with memory management; and
- continue our work on formalizing hardware interfacing and use specifications.

This last item is hardest and has the biggest payoff.
Industrial Collaborations

We have been working with DEC for two years.

Motorola may attempt the specification (and possibly the verification) of one of their microcontrollers.

Technology Transfer

We highly value interactions with industry; we all profit.

Our formal techniques may be used incrementally, i.e., "creeping formalization."

Industry first employs our techniques for (unambiguous) specification, later for verification.

Specification is a big problem for industry -- formal specification allows analysis without exhaustive testing.
Technology Enablers

Is the state-of-the-art separating further from the state-of-the-practice?

To enable the use of formal techniques in hardware design we need to:

- train more engineers with formal methods (not train mathematicians to be engineers);
- make existing tools and techniques more accessible to engineers; and
- make formal techniques the most economical method of hardware validation.

A big success or two would help us get industry's attention.
Conclusions

Formal methods can be used to provide accurate specifications.

Hardware verification provides increased assurance of circuit correctness.

Formal techniques provide a good growth path; they scale up well.

The credit rating of formal techniques is improving.

Goals:

• Completely replace programmer's manuals, timing diagrams, interface specifications, power requirements, etc. with clear precise formulas.

• Provide a perfectly clear foundation upon which systems can be built.
Generic Interpreters
and
Microprocessor Verification

Phillip J. Windley

Department of Computer Science
University of Idaho

August, 1990

This work was sponsored under Boeing Contract NAS1-18586, Task Assignment No. 3, with NASA-Langley Research Center.
Outline

- Introduction
- Generic interpreters
- Microprocessor Verification
- Future Work
Microprocessor Verification

- VIPER, the first commercially available, "verified" microprocessor, has never been formally verified.

- The proof was not completed even though 2 years were spent on the verification.
Microprocessor Verification
(continued)

• Our research is aimed at making the verification of large microprocessors tractable.

• Our objective is to provide a framework in which a masters–level student can verify VIPER in 6 person–months.
Determining Correctness

In VIPER (and most other microprocessors), the correctness theorem was shown by proving that the electronic block model implies the macro–level specification.
The Problem
(continued)

- Microprocessor verification is done through case analysis on the instructions in the macro level.

- The goal is to show that when the conditions for an instruction's selection are right, the electronic block model implies that it operates correctly.

- A lemma that the EBM correctly implements each instruction can be used to prove the top-level correctness result.
The Problem

Unfortunately, the one-step method doesn't scale well because

- The number of cases gets large.

- The description of the electronic block model is very large.
Hierarchical Decomposition

- A microprocessor specification can be decomposed hierarchically.

- The abstract levels are represented explicitly.
Interpreters

An abstract model of the different layers in the hierarchy provides a methodological approach to microprocessor verification.

- The model drives the specification.

- The model drives the verification.
Interpreters
(top level)
Specifying an Interpreter
(overview)

We specify an interpreter by:

- Choosing a $n$–tuple to represent the state, $S$.
- Defining a set of functions denoting individual interpreter instructions, $J$.
- Defining a next state function, $N$.
- Defining a predicate denoting the behavior of the interpreter, $I$. 
Verifying an Interpreter
(overview)

We verify an interpreter, I with respect to its implementation M by showing

\[ M \Rightarrow I. \]

To do this, we will show that every instruction in J can be correctly implemented by M:

\[ \forall j \in J. \quad M \Rightarrow (\forall t: \text{time. } \quad C(t) \Rightarrow s(t + n) = j(s(t))) \]

where \( C \) represents the conditions for instruction \( j \)'s selection.
We have designed and are verifying a microcomputer with interrupts, supervisory modes and support for asynchronous memory.

- The datapath is loosely based on the AMD 2903 bit-sliced datapath.

- The instruction format is very simple.

- The control unit is microprogrammed.
AVM-1’s Instruction Set
(subset)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>JMP</td>
<td>jump on 16 conditions</td>
</tr>
<tr>
<td>000001</td>
<td>CALL</td>
<td>call subroutine</td>
</tr>
<tr>
<td>000010</td>
<td>INT</td>
<td>user interrupt</td>
</tr>
<tr>
<td>000110</td>
<td>LD</td>
<td>load</td>
</tr>
<tr>
<td>000111</td>
<td>ST</td>
<td>store</td>
</tr>
<tr>
<td>010000</td>
<td>ADD</td>
<td>add (3-operands)</td>
</tr>
<tr>
<td>011011</td>
<td>SUBI</td>
<td>subtract immediate (2-operands)</td>
</tr>
<tr>
<td>011111</td>
<td>NOOP</td>
<td>no operation</td>
</tr>
</tbody>
</table>

- The architecture is load-store.
- The instruction set is RISC-like.
- There is a large register file.
Figure 5.2: The AVM-1 Datapath
The Phase–Level Specification

The n–tuple representing the state:

\[ S_{\text{phase}} = (\text{mir, mpc, reg, alatch, blatch, mar, mbr, clk, mem, urom, ireq, iack}) \]
The Phase-Level Specification

A typical function specifying an instruction's behavior from $J_{\text{phase}}$:

$$
\text{\texttt{def phase_two rep}} \ (\texttt{mir}, \texttt{mpc}, \texttt{reg}, \texttt{alatch}, \texttt{blatch}, \texttt{mbr}, \texttt{mar}, \texttt{clk}, \texttt{mem}, \texttt{urom}, \texttt{ireq}, \texttt{iack}) =

\texttt{(mir, mpc, reg,}
\texttt{ EL (bt5_val (SrcA mir)) \ reg,}
\texttt{ EL (bt5_val (SrcB mir)) \ reg,}
\texttt{ mbr, mar, (T,F), mem, urom, ireq, Iack mir)}
$$
The Electronic Block Model

The electronic block model is not specified as an interpreter.

- EBM is a *structural* specification.

- The specification
  - is in terms of smaller blocks.
  - uses existential quantification to hide internal lines.
Objects

There are several abstract classes of objects that we will use to define and verify an abstract interpreter.

*:state An object representing system state.

*:key The identifying tokens for instructions.

*time A stream of natural numbers.

We will prime class names to indicate that the objects are from the implementing level.
Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst_list</td>
<td>((\astkey \times (\aststate \rightarrow \aststate))\text{list})</td>
</tr>
<tr>
<td>key</td>
<td>(\astkey \rightarrow \text{num})</td>
</tr>
<tr>
<td>select</td>
<td>(\aststate \rightarrow \astkey)</td>
</tr>
<tr>
<td>cycles</td>
<td>(\astkey \rightarrow \text{num})</td>
</tr>
<tr>
<td>substate</td>
<td>(\aststate' \rightarrow \aststate)</td>
</tr>
<tr>
<td>Impl</td>
<td>((\text{time} \rightarrow \aststate') \rightarrow \text{bool})</td>
</tr>
<tr>
<td>clock</td>
<td>(\aststate' \rightarrow \astkey')</td>
</tr>
<tr>
<td>begin</td>
<td>(\astkey')</td>
</tr>
</tbody>
</table>
The *instruction correctness lemma* is important in the generic interpreter verification.

Here is the generic version of that lemma for a *single* instruction:

\[
\Gamma \overset{\text{def}}{=} \text{INST-CORRECT } s' \text{ inst } = \\
(\text{Impl } s') \Rightarrow \\
\forall t' : \text{time}'.
\]

let \( s = (\lambda t. \text{substate}(s' t')) \) in

let \( c = (\text{cycles}(\text{select}(s t'))) \) in

(\text{select}(s t') = (\text{FST inst})) \land \\
(\text{clock}(s' t') = \text{begin}) \Rightarrow \\
((\text{SND inst}) (s t') = (s(t' + c))) \land \\
(\text{clock}(s'(t' + c)) = \text{begin})
Using the predicate INST_CORRECT, we can define the theory obligations:

1. The *instruction correctness lemma*:

   \[ \text{EVERY } (\text{INST\_CORRECT } s') \text{ inst\_list} \]

2. Every key selects an instruction:

   \[ \forall k : \text{*key}. (\text{key } k) < (\text{LENGTH inst\_list}) \]

3. The instruction list is ordered correctly:

   \[ \forall k : \text{*key}. k = (\text{FST (EL (key } k \text{) inst\_list)}) \]
Generic Interpreters
Instantiation

- Generic Interpreter + Macro Level Definitions → Macro Level Interpreter
- Generic Interpreter + Micro Level Definitions → Micro Level Interpreter
- Generic Interpreter + Phase Level Definitions → Phase Level Interpreter

Electronic Block Model
Interpreter Theory
(temporal abstraction)

We need to show a relationship between the state stream at the implementation level and the state stream at the top level.

\[ \begin{array}{c}
\begin{array}{ccccccccc}
& t_1 & & t_2 & & t_3 & & t_4 & & t_5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\end{array} \]

The function \( f \) is a temporal abstraction function for streams.
An interpreter's behavior is specified as a predicate over a state stream.

\[ \vdash_{\text{def}} \text{INTERP} \ s = \]
\[ \forall t : \text{time}. \]
\[ \text{let } n = (\text{key}(\text{select}(s \ t))) \ \text{in} \]
\[ s(t + 1) = (\text{SND} \ (\text{EL} \ n \ \text{inst_list}))(s \ t) \]
Interpreter Theory
(correctness result)

Our goal is to verify an interpreter, \( I \) with respect to its implementation \( M \) by showing

\[ M \Rightarrow I. \]

Here is the abstract result:

\[ \vdash \text{Impl } s' \land (\text{clock}(s' 0) = \text{begin}) \Rightarrow \text{INTERP } (s \circ f) \]

where

\[ s = (\lambda t : \text{time. substate}(s' t)) \quad \text{and} \]
\[ f = (\text{time_abs } (\text{cycles} \circ \text{select})s) \]
Instantiating a Theory

Instantiating the abstract interpreter theory requires:

- Defining the abstract constants.
- Proving the theory obligations.
- Running a tool in the formal theorem prover.
Definitions

We wish to instantiate the abstract interpreter theory for the phase-level. The electronic block model will be the implementing level.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instantiation</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst_list</td>
<td>a list of instructions</td>
</tr>
<tr>
<td>key</td>
<td>bt2_val</td>
</tr>
<tr>
<td>select</td>
<td>GetPhaseClock</td>
</tr>
<tr>
<td>cycles</td>
<td>PhaseLevelCycles</td>
</tr>
<tr>
<td>substate</td>
<td>PhaseSubstate</td>
</tr>
<tr>
<td>Impl</td>
<td>EBM</td>
</tr>
<tr>
<td>clock</td>
<td>GetEBMClock</td>
</tr>
<tr>
<td>begin</td>
<td>EBM_Start</td>
</tr>
</tbody>
</table>
An Example

After proving the theory obligations, we can perform the instantiation.

```plaintext
let theorem_list =
  instantiate_abstract_theorems
  'gen_I'
  [Phase_I_EVERY_LEMMA;
   Phase_I_LENGTH_LEMMA;
   Phase_I_KEY_LEMMA]
  [
    "([(F,F),phase_one;
       (F,T),phase_two
       (T,F),phase_three
       (T,T),phase_four],
      bt2_val, GetPhaseClock,
      PhaseLevelCycles, PhaseSubstate,
      EBM, GetEBMClock, EBM_Start)";
    "(λ t:time. (mir t, mpc t, reg_list t,
                alatch t, blatch t,
                mbr_reg t, mar_reg t,
                clk t, mem t, urom))"
  ]
  'PHASE';
```

73
The Electronic Block Model

\[ EBM \text{ rep } (\lambda t. (\text{mir } t, \text{mpc } t, \text{reg } t, \text{alatch } t, \text{blatch } t, \text{mbr } t, \text{mar } t, \text{clk } t, \text{mem } t, \text{urom}, \text{ireq } t, \text{iack } t)) = \]

\[ \exists \text{opc } \text{ie_s } \text{sm_s } \text{iack_s } \text{amux_s } \text{alu_s } \text{sh_s } \text{mbr_s } \text{mar_s } \text{rd_s } \text{wr_s } \text{cselect } \text{bselect } \text{aselect } \text{neg_f } \text{zero_f } (\text{float:time->bool}). \]

\[ \text{DATAPATH rep } \text{amux_s } \text{alu_s } \text{sh_s } \text{mbr_s } \text{mar_s } \text{rd_s } \text{wr_s } \text{cselect } \text{bselect } \text{aselect } \text{neg_f } \text{zero_f } \text{float } \text{float } \text{ireq } \text{iack_s } \text{iack_opc } \text{ie_s } \text{sm_s } \text{clk } \text{mem } \text{reg } \text{alatch } \text{blatch } \text{mar_reg } \text{mbr_reg } \text{reset_e } \text{ireq_e } \wedge \]

\[ \text{CONTROL_UNIT rep } \text{mpc } \text{mir } \text{clk } \text{amux_s } \text{alu_s } \text{sh_s } \text{mbr_s } \text{mar_s } \text{rd_s } \text{wr_s } \text{cselect } \text{bselect } \text{aselect } \text{neg_f } \text{zero_f } \text{ireq } \text{iack_s } \text{opc } \text{ie_s } \text{sm_s } \text{urom } \text{reset_e } \text{ireq_e } \]

Fully expanded, the electronic block model specification fills about six pages.
Future Work

- New architectural features.
- Composing verified blocks.
- Verifying operating systems.
- Gate-level verification.
- Byte-code interpreter verification.
- Other classes of computer systems.
An Example
(continued)

After some minor manipulation, the final result becomes:

\[ \vdash \text{EBM} \]
\[
(\lambda t.
  (\text{mir } t, \text{mpc } t, \text{reg_list } t, \text{alatch } t, \text{blatch } t,
  \text{mbr_reg } t, \text{mar_reg } t, \text{clk } t, \text{mem } t, \text{urom}) \Rightarrow
\text{Phase_I})
\]
\[
(\lambda t.
  (\text{mir } t, \text{mpc } t, \text{reg_list } t, \text{alatch } t, \text{blatch } t,
  \text{mbr_reg } t, \text{mar_reg } t, \text{clk } t, \text{mem } t, \text{urom}))
\]
Conclusions

The generic proof

- Cleared away all the irrelevant detail.

- Formalized the notion of interpreter proofs which has been used in several microprocessor verifications.

- Provided a structure for future microprocessor verifications.
The VIPER project has so far produced a formal specification of a 32 bit RISC microprocessor, an implementation of that chip in radiation-hard SOS technology, a partial proof of correctness of the implementation which is still being extended, and a large body of supporting software. The time has now come to consider what has been achieved and what directions should be pursued in future.

The most obvious lesson from the VIPER project has been the time and effort needed to use formal methods properly. Most of the problems arose in the interfaces between different formalisms e.g. between the (informal) English description and the HOL spec, between the block-level spec in HOL and the equivalent in ELLA needed by the low-level CAD tools. These interfaces need to be made rigorous or (better) eliminated.

VIPER 1A (the latest chip) is designed to operate in pairs, to give protection against breakdowns in service as well as design faults. We have come to regard redundancy and formal design methods as complementary, the one to guard against normal component failures and the other to provide insurance against the risk of the common-cause failures which bedevil reliability predictions.

Any future VIPER chips will certainly need improved performance to keep up with increasingly demanding applications. We have a prototype design (not yet specified formally) which includes 32 and 64 bit multiply, instruction pre-fetch, more efficient interface timing, and a new instruction to allow a quick response to peripheral requests. Work is under way to specify this device in MIRANDA, and then to refine the spec into a block-level design by top-down transformations. When the refinement is complete, a relatively simple proof checker should be able to demonstrate its correctness.
Example of NODEN output

The NODEN analysis suite provides automatic comparison between the specification and design of moderately complex blocks of logic. The following example is taken from the VIPER design. MINOR is the simplest block in the chip, essentially consisting of a three bit counter. Following this paragraph is its specification in NODEN-HDL, whilst on the following pages are a correct and incorrect implementation. The final page shows the output of the comparison program when presented with the erroneous circuit.

\** MINOR STATE LOGIC in NODEN **\

FN INCWORD3 = (word3: minor) -> word3:
   IF (VAL3 minor) = 7
      THEN WORD3 0
   ELSE WORD3((VAL3 minor)+1)
   FI.

BLOCK MINOR = (bool: nextmainbar advance
              reset intresetbar)
       -> (~word3: minor):
   IF reset OR (NOT intresetbar) OR
      (advance AND (NOT nextmainbar))
      THEN WORD3 0
   ELIF advance
      THEN INCWORD3 minor
   ELSE minor
   FI.
\ *** Library' of primitive gate functions *** \\

FN INV = (bool: a ) -> bool: NOT a.
FN NAND2= (bool: a b ) -> bool: NAND(a,b).
FN EXNOR = (bool: a b ) -> bool: a = b.
FN ORNAND= (bool: a b c d) -> bool: NAND(a OR b,c OR d).

\ NB. NAND3 & NAND4 are built-in functions \\

\ *** Correct gate level implementation *** \\

BLOCK MINOR = (bool: nextmnbar advance reset intrstbar)
            -> (^word3: minor):
            BEGIN
                LET qbar_1 := NOT (minor[1]),
                      qbar_2 := NOT (minor[2]),
                      qbar_3 := NOT (minor[3]).

                LET gb2 := INV(advance).
                LET gb4 := INV(reset).
                LET gb1 := NAND4(nextmnbar,advance,gb4,intrstbar).
                LET gb3 := NAND3(gb2, gb4, intrstbar).
                LET gb7 := INV(qbar_1).
                LET gb8 := EXNOR(qbar_1, qbar_2).
                LET gb11 := INV(qbar_2).
                LET gb12 := NAND2(gb7, gb11).
                LET gb13 := EXNOR(gb12, qbar_3).

                OUTPUT (ORNAND(gb7, gb1, gb3, qbar_1),
                        ORNAND(gb8, gb1, gb3, qbar_2),
                        ORNAND(gb13, gb1, gb3, qbar_3)
                        )
            END.
\*\*\* Wrong gate level implementation \*\*\* \\

BLOCK M_ERR = (bool: nextmnbar advance reset intrstbar) 
-> (^word3: minor):
BEGIN
    LET qbar_1 := NOT (minor[1]),
            qbar_2 := NOT (minor[2]),
            qbar_3 := NOT (minor[3]).

    LET gb2 := INV(advance).
    LET gb4 := INV(reset).
    LET gb1 := NAND4(nextmnbar, advance, gb4, intrstbar).
    LET gb3 := NAND3(gb2, gb4, intrstbar).
    LET gb7 := INV(qbar_1).

    \** Inverted qbar_2 ** \n    LET gb8 := EXNOR(qbar_1, NOT qbar_2).

    LET gb11 := INV(qbar_2).

    \** Missing NAND with gb7 ** \n    LET gb12 := gb11.

    LET gb13 := EXNOR(gb12, qbar_3).

    \** Inverted first output ** \n    OUTPUT (NOT(ORNAND(gb7, gb1, gb3, qbar_1)),
            ORNAND(gb8, gb1, gb3, qbar_2),
            ORNAND(gb13, gb1, gb3, qbar_3))
END.
Specification: 'MINOR' Implementation: 'M_ERR'

COMPARISON ERROR: Implementation output 'minor[1]' is always incompatible with the specification of 'minor[1]'; output inverted?

COMPARISON ERROR: Implementation output 'minor[2]' is incompatible with the specification of 'minor[2] under the following circumstances:

\[
\begin{align*}
\text{nextmainbar} &= t \\
\text{advance} &= t \\
\text{reset} &= f \\
\text{intresetbar} &= t
\end{align*}
\]

For specification output 'minor[3]'' implementation output 'minor[3]':

WARNING: Specification depends on minor[1] and implementation doesn't

COMPARISON ERROR: Implementation output 'minor[3]' is incompatible with the specification of 'minor[3] under the following circumstances:

\[
\begin{align*}
\text{nextmainbar} &= t \\
\text{advance} &= t \\
\text{reset} &= f \\
\text{intresetbar} &= t \\
\text{minor[2]} &= f
\end{align*}
\]

*** Comparison fails, invalid implementation ***
NODEN changes

- Negative integer subranges allowed
  E.g. TYPE i8 = INT[-128..127].

- Automatic casts between types
  E.g. (t,t,f) + bool3_val + i8_val

- 2's compliment []bool to integer ops.

- Explicit legal value, !bool

- Compiler about four times faster.

- Analyer about twice as fast.
Old NODEN_HDL

FN INCWORD3 = (word3: minor) -> word3:
    IF (VAL3 minor) == 7
        THEN WORD3 0
    ELSE WORD3 ((VAL3 minor) + 1)
    FI.

New NODEN_HDL

FN INCWORD3 = (word3: minor) -> word3:
    IF minor == 7 THEN 0 ELSE minor + 1 FI.
Bibliography


Why VIPER2?

- Faster, 32 and 64 bit multiply
- Improved interface to outside world
- New design methods now available
Extra speed by ..

- Instruction pre-fetch
- Dedicated adders for P and indexing
- Half-cycle overlaps rather than full cycle

Speed more than 3x at same clock frequency
On-board Multiply Instructions

Three separate instructions, $F = 13, 14, 15$

- Signed, 32 bit product, stop on OVF
- Unsigned, LS 32 bits of product
- Unsigned, MS 32 bits of product
Improved interface

- "Call on signal" instruction

- "Frame restart" input

- Longer setup and hold times on memory and I/O cycles
New design methods

Top-down synthesis by correctness-preserving transformations

• Starts from specification in MIRANDA

• Generates proof as part of design process

• May scale up better than post hoc proof
VIPER 1A perspective

The present chip falls in between the main application areas:

- Automotive and comms: too expensive, minimum system too big (5 memory chips)

- Avionics: not fast enough, no multiply

- Space: about right, tiny market
Processor 1

Exactly Equal? [Yes → OPERATE, No → STOP]

Processor 2
Dependable

Error

Reporting
Mechanical Proofs of Fault-tolerant Clock Synchronization

N. Shankar

Computer Science Laboratory
SRI International
Overview

Introduction to clock synchronization protocols?

A schematic formulation of clock synchronization (Schneider).

The Interactive Convergence Algorithm (Lamport/Melliar-Smith).

Verification of Schneider's formulation (Shankar).

Verification of Interactive Convergence (Rushby/von Henke).

A hardware-oriented clock synchronization protocol (Infis/Moore).

Verification of Infis/Moore's protocol (Rushby/Shankar).

The EHDM Specification/Verification Environment.

Conclusions.
Main Observations

- Fault-tolerant clock synchronization is a critical component of a real-time control system.

- Proofs of the correctness of clock synchronization are complex and subtle.

- Informal proofs tend to be tenuous in these domains.

- Formal verification is a useful way to reduce errors and achieve reliable designs.

- Specification/Verification could contribute to the scientific foundations of reliable engineering.
Fault-tolerant systems

- Critical real-time control systems such as "fly-by-wire" digital avionics.

- Replicated processors are used to provide hardware fault-tolerance.

- Results are periodically voted.

- Clocks must be synchronized to ensure approximately synchronous behaviour across nonfaulty processors.
Clock Synchronization

- Clocks start synchronized.

- Over time, the clocks drift apart.

- The clocks are periodically synchronized by
  - an exchange of clock values
  - computation of a mutually agreeable clock value
  - adjustment of the logical clock
Byzantine Clocks

Three clocks $A$, $B$, $C$.

Suppose clocks drift away from real time by upto a minute an hour.

$C$ is faulty.

Clocks resynchronize around noon and exchange clock values.

$A$ reads 12:00 and $B$ reads 11:59

$A$ transmits 12:00 to $B$ and $C$.

$B$ transmits 11:59 to $A$ and $C$.

$C$ maliciously transmits 12:01 to $A$; 11:58 to $B$. 
Byzantine Clocks

Three clocks \( A, B, C \).

Clocks drift from real time by up to a minute an hour.

\( C \) is faulty.

Clocks resynchronize around noon and exchange clock values.

\( A \) reads 12:00 and \( B \) reads 11:59

\( A \) resets its clock to the mean of the acceptable clock values, i.e., 12:00.
\( B \) similarly resets itself to 11:59.

\( A \) and \( B \) are not any closer following resynchronization.
Clock Generalities

No global clocks — single point of failure, therefore not fault-tolerant.

Synchronization is with respect to other clocks, not real time, though such protocols do exist.

Clocks drift at rate $\rho$ with respect to real time.

Period of drift $R$ between resynchronization rounds.

$\epsilon$ bounds the error in reading clock values.

To keep clocks synchronized to within $\delta$, clocks should be within $\delta_s$ following resynchronization, and

$$\delta > \delta_s + 2\rho R$$

Each clock uses the same convergence function to synchronize to within $\delta_s$. 
## Typical numbers (from Rushby/von Henke)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>6</td>
<td>No. of Clocks</td>
</tr>
<tr>
<td>$R$</td>
<td>104.8 msec.</td>
<td>Period</td>
</tr>
<tr>
<td>$\delta_0$</td>
<td>132 $\mu$sec.</td>
<td>Initial skew</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>66.1 $\mu$sec.</td>
<td>Reading error</td>
</tr>
<tr>
<td>$\rho$</td>
<td>$15 \times 10^{-6}$</td>
<td>Drift rate</td>
</tr>
<tr>
<td>$\delta$</td>
<td>271 $\mu$sec. ($F = 1$)</td>
<td>Maximum skew</td>
</tr>
</tbody>
</table>
Clock Requirements

- R1: At any instant, two nonfaulty clock readings should be no further than $\delta$ apart.

- R2: There should be a small bound on the adjustment needed to resynchronize a clock.
Schneider’s Schema

A generalization of various protocols consisting of:

- Assumptions on the behavior of nonfaulty physical clocks.

- Constraints on the computation of nonfaulty logical clocks.

These assumptions and constraints are used to derive a bound on the skew between two nonfaulty logical clocks, i.e.

\[ |LC_p(t) - LC_q(t)| \leq \delta \]
Physical Clock Assumptions

$N$ clocks with at most $F$ faulty.

t$_p^i$ is the time at which $p$ resets its clock for the $i$'th time.

Interval between resets is bounded:

$$r_{min} \leq t_{p}^{i+1} - t_{p}^{i} \leq r_{max}$$

Skew between resets is bounded: $|t_{p}^{i} - t_{q}^{i}| \leq \beta$

Bounded drift rate w.r.t. real time: for $s > t$

$$(s - t)(1 - \rho) \leq C_p(s) - C_p(t) \leq (s - t)(1 + \rho)$$
Logical Clock Assumptions

A Convergence function $Cfn$ is used to compute the adjusted logical clock.

Let $\Theta_p^i(q)$ be $p$'s reading (estimate) of $q$'s logical clock at time $t_p^i$.

Then $LC_p(t_p^i) = Cfn(p, \Theta_p^i)$

The $i$'th adjustment to be applied to the physical clock to derive the logical clock is

$$Adj_p^i = Cfn(p, \Theta_p^i) - C_p(t_p^i)$$

In general the logical clock is defined to be

$$LC_p(t) = C_p(t) + Adj_p^i$$

for $t_p^i \leq t < t_p^{i+1}$

$\epsilon$ bounds error with which clocks are read.

Additionally, certain assumptions on behavior of a satisfactory convergence function.
Translation Invariance

Adding $X$ to each clock reading, adds $X$ to the value of the convergence function.

For any $X$ and $\theta$ mapping clock numbers to clock readings

$$Cfn(p, (\lambda q:\theta(q) + X)) = Cfn(p, \theta) + X$$

Translation invariance is used to compare the values of convergence functions at $t_p^i$ and $t_q^i$. 
Precision Enhancement

Formalizes the intuition that

- the closer the good clocks are to each other

- the closer the different readings of the same good clock

- then the closer the resulting convergence function values
Precision Enhancement (contd.)

Given any predicate $P$ on clocks 0 to $N - 1$ that holds of at least $N - F$ clocks.

Given $p, q$, such that $P(p)$ and $P(q)$.

Given $\theta_p$ and $\theta_q$ such that

- If $P(l)$ and $P(m)$, then $|\theta_p(l) - \theta_p(m)| \leq Y$
- If $P(l)$ and $P(m)$, then $|\theta_q(l) - \theta_q(m)| \leq Y$
- If $P(l)$, then $|\theta_p(l) - \theta_q(l)| \leq X$

Then there exists a bound $\pi(X, Y)$ such that

$$|Cfn(p, \theta_p) - Cfn(q, \theta_q)| \leq \pi(X, Y)$$

Illustrative example to follow.
Accuracy Preservation

Bounds the adjustment away from a good clock reading.

Given any predicate $P$ on clocks 0 to $N - 1$ that holds of at least $N - F$ clocks.

Given that $P$ holds of $p$ and $q$.

Given $\theta_p$ such that whenever $P(l)$ and $P(m)$ for any two clocks $l$ and $m$, then

$$|\theta_p(l) - \theta_p(m)| \leq Z$$

Then

$$|Cf_n(p, \theta_p) - \theta_p(q)| \leq \alpha(Z)$$

That is, if the good clock readings are within $Z$, the adjustment away from a good clock reading is no more than $\alpha(Z)$. 
The Final Result: Agreement

- **A1**: $\beta \leq r_{\min}$
  Synchronization rounds are distinct

- **A2**: $\delta_0 \leq \delta_s$
  Initial skew no greater than skew immediately following synchronization.

- **A3**: $\delta_s + 2\rho r_{\max} \leq \delta$
  Drift between synchronization rounds is below $\delta$.

- **A4**: $\pi(2\epsilon + 2\rho\beta, \delta_s + 2\rho(r_{\max} + \beta) + 2\epsilon) \leq \delta_s$
  Skew between just synchronized clocks below $\delta_s$.

- **A5**: $\alpha(\delta_s + 2\rho(r_{\max} + \beta) + 2\epsilon) \leq \delta$
  Skew between synchronized and yet to be synchronized clocks below $\delta$. 

20
• Conclusion:

\[
\begin{align*}
t \geq 0 \\
\land \; \text{correct}(p, t) \\
\land \; \text{correct}(q, t) \\
\Rightarrow \; |LC(p, t) - LC(q, t)| \leq \delta
\end{align*}
\]

Skew between nonfaulty logical clocks bounded by \(\delta\).
Verification of Schneider's Schema using EHDM

Proof consists of:

- 30 axioms involving multiplication, division, and clocks.
- 12 definitions
- 95 lemmas.

Proof took about two man-months using EHDM. Machine verification takes 1000 to 3500 CPU secs on SUNs.

Numerous inaccuracies in Schneider's original presentation were corrected.

The machine proof adds enormous clarity to Schneider's insightful, but imprecise descriptions and definitions.

Instantiation of Schneider's schema in progress.
Lamport/Melliar-Smith's Interactive Convergence (ICA)

$3F + 1$ clocks needed to tolerate $F$ Byzantine faults.

$p$ records (relative discrepancies of) other clock values when its clock reads $iR$.

"Ignores" clock readings further than $\Delta$ away.

Adjusts its clock by the 'egocentric' mean of the acceptable clock differences.
**Instantiating Schneider’s protocol with ICA**

Convergence function:

$$ica(p, \theta) = \sum_{l=0}^{N-1} \frac{fix_p(\theta(l), \theta)}{N}$$

where

$$fix_p(x, \theta) = \begin{cases} \begin{array}{ll} x & \text{if } |x - \theta(p)| \leq \Delta \\ \theta(p) & \text{otherwise} \end{array} \end{cases}$$

Translation Invariance: Note that

$$fix_p((\lambda l : \theta(l) + t)(q)_\theta) = (\theta(q)\theta + t)$$
Precision Enhancement of ICA

Given that for all correct $l, m$

- $|\theta_p(l) - \theta_q(l)| \leq X$
- $|\theta_p(l) - \theta_p(m)| \leq Y$
- $|\theta_q(l) - \theta_q(m)| \leq Y$

We have

\[
|\text{ica}(p, \theta_p) - \text{ica}(q, \theta_q)| \\
\leq X + \frac{FY + 2F\Delta}{N} \\
= \pi(X, Y)
\]

$X$ is negligible, but $Y \approx \Delta$, so

\[
\pi(X, Y) \approx \frac{3F\Delta}{N}
\]

Since $\Delta \geq \delta + \epsilon$, we get $N > 3F + 1$. 

Accuracy Preservation of ICA

If nonfaulty clock readings are Z apart, then F faulty clocks can contribute a further skew of \( F\Delta/N \) to the egocentric mean.

So

\[
\alpha(Z) \leq Z + \frac{F\Delta}{N}
\]
Rushby/von Henke's verification of ICA using EHDM

Around 1–2 man month effort

20 modules

1,550 lines of specification

166 proofs

1 hour elapsed to prove them all on Sun 3/75-8

Verification revealed several minor flaws in a five year old journal proof.
Flaws in Lamport/Melliar-Smith

Main induction incorrect (bad approximations)

Proof of Lemma 4 incorrect (bad approximations); also typographical error in statement

Lemma 1 false in absence of additional constraints in A2

Lemma 2 similarly, also typographical error in statement

Lemma 3 similarly, and unnecessarily general

Missing requirement for S2 in Lemmas 1, 3, 4, and (when repaired) 2
Original Constraints on parameters

C1:

C2:

C3: \( \Sigma = \Delta \)

C4: \( \Delta \geq \delta + \epsilon \)

C5: \( \delta \geq \delta_0 + \rho R \)

C6: \( \delta \geq 2(\epsilon + \rho S) + \frac{2m\Delta}{n-m} + \frac{n\rho R}{n-m} \)
New Constraints on parameters

**C1:** \( R \geq 3S \)

**C2:** \( S \geq \Sigma \)

**C3:** \( \Sigma \geq \Delta \)

**C4:** \( \Delta \geq \delta + \epsilon + \frac{\rho}{2} S \)

**C5:** \( \delta \geq \delta_0 + \rho R \)

**C6:**
\[
\delta \geq 2(\epsilon + \rho S) + \frac{2m\Delta}{n-m} + \frac{n\rho R}{n-m} + \frac{n\rho \Sigma}{n-m} + \rho \Delta
\]
Infis/Moore's economic approach

Tolerates $F < N/2$ omission failures for $N$ clocks.

At clock reading $iR$, $p$ broadcasts a pulse on its private line.

Say $p$ receives and validates $N - f$ pulses

$(N - F)$'th pulse bounded from above and below by a good pulse.

Ditto for $(F - f + 1)$'th pulse.

$p$ starts new clock at earlier of pulse $N - F$ with delay $D$, or pulse $F - f + 1$ with delay $2D$.

Skew $\delta_s \lesssim D$, and $\delta \lesssim 2D$.

Verification nearly complete using EHDM. Elaborates significantly on informal proof.
Schemata for Infis/Moore's protocol
Extract from Infis/Moore

(a) \( T^k_{n-i} \geq T^k_{n-i} \) because the \( T^k_i \) are a subset of the \( T^k \).
(b) \( T^k_{n-i} \leq T^k_{n-m} \) because at least one of the times \( T^k_{n-i} \) ... \( T^k_{n-f} \) must be a message from a processor which is actually fault-free (and synchronised) and \( T^k_{n-m} \) is either the time of the message from the last fault-free processor or later.
(c) \( T^k_{n-f} \geq T^k_{n-m} \) because the \( T^k_{n-m} \) is validated by all fault-free processors and must be included in the \( T^k_i \).
(d) \( T^k_{n-f} \leq T^k_{n-g} \) because the \( T^k_i \) are a subset of the \( T^k \).

From these inequalities we have that

\[
\min \{ T^k_{n-i} + d, T^k_{n-m} \} \leq W \leq \min \{ T^k_{n-m} + d, T^k_{n-g} \} \tag{1}
\]

Now \( T^k_{n-f+1} \leq T^k_{n-i} \) for all \( k \) and \( T^k_{n-f} = T^k_{n-g} \) for some \( k \), so the validity tests \( T^k_{n-f} - T^k_{n-f+1} < 2d \) imply that \( T^k_{n-g} - T^k_{n-i} < 2d \). Therefore \( T^k_{n-m} - T^k_{n-i} < d \) or \( T^k_{n-g} - T^k_{n-m} < d \) (or both).

If \( T^k_{n-m} - T^k_{n-i} < d \), eqn. 1 reduces to

\[
T^k_{n-m} \leq W \leq \min \{ T^k_{n-m} + d, T^k_{n-g} \}
\]

implying that \( W \) has a range of at most \( d \).

If \( T^k_{n-g} - T^k_{n-m} < d \), then, using also that \( T^k_{n-g} - T^k_{n-i} < 2d \), eqn. 1 yields

\[
T^k_{n-g} - d < W \leq T^k_{n-g}
\]

implying that \( W \) has a range less than \( d \).
Verification of Infis/Moore's protocol

Formalization is fairly close to hardware realization.

Main induction over synchronization rounds completed, as well as all of the important lemmas.

Machine proof is remarkably involved and complex.

Proof took two man-months of effort and covers about 70 dense pages.
Common Errors

Ignoring failures.

Distinguishing real and clock time, and relative versus absolute measurements.

Ignoring small but significant quantities.

Proving one statement but using another.

Imprecise definitions.

Erroneous algebraic manipulations.

Implicit assumptions.

Incorrect assumptions.
Difficulties in verification

Dealing simultaneously with failures, temporal ordering, relative measurements, drift.

Have to be careful not to assume anything about failed clocks.

"Circular definitions" need to be avoided. E.g., A round ends when various events have taken place. Various events take place as scheduled if the clock is correct at the end of the round.

Mentally retaining all the relevant facts is difficult.
EHDM specification/verification system

Based on a simply typed higher-order logic with subtyping.

Parametric modules used to structure specifications.

Specifications can be proved to implement other specifications.

Components include parser, typechecker, theorem prover, Hoare sentence prover, and MLS tool.

Theorem prover contains powerful decision procedures for integer and rational inequalities.

New implementation should be ready by end of 1990.
Concluding Observations

Reasoning about fault-tolerant clock synchronization is extremely difficult.

Proofs involve heavy use of inequalities, algebraic manipulations, finite set theory, and induction.

Protocol designers themselves feel the need for mechanized verification tools.

Benefits of such tools are:

- Design discipline
- Efficient location/correction of design errors
- Design library for future reuse
- Standardized language for communicating designs and proofs

Specification and verification technology could contribute effectively to the foundations of reliable engineering.
A HOL Theory for Voting

Paul S. Miner       James L. Caldwell
Outline

• Introduction
• Proofs Comparing Majority and Plurality
• Proofs of Simple Reconfiguration Strategies
• Future directions
Introduction

- Central to fault-tolerant computing is redundancy management.

- Common to proofs of fault-tolerance is a maximum fault assumption.

  If there are \( m \) or fewer faults in the system, then . . .

- Typically a maximum fault assumption is rather restrictive. Usually, this is necessary to avoid assumptions about the behavior of faulty channels.

  - For Interactive consistency, in order to tolerate \( m \) faults, \( 3m + 1 \) nodes are required.
  
  - For a majority vote, \( 2m + 1 \) channels are required.

- A maximum fault assumption is useful because it allows us to reason about fault tolerance in the presence of arbitrarily malicious fault behavior. However, analysis of the architecture may establish certain scenarios in which the assumption may be weakened.
• Should fault-tolerant systems incorporate features which attempt to recover from failure combinations which exceed the maximum fault assumption?

• If so, what is the proof obligation?

• At the very least, it is necessary to show that existing proofs which depend upon the maximum fault assumption still hold.
Hypothetical Scenario

Imagine that plurality voting circuit has been developed for use in a four channel fault-tolerant computing system. Suppose that a designer is considering using this circuit in a system which depends upon a majority vote in order to maintain correct system state.

Can this voting circuit be used in this system?
First we define existence predicates for majority and plurality as follows:

\[ \forall B. \text{majority\_exists } B \equiv \text{FINITE } B \land \exists x. |B| < 2|B|_x \]
\[ \forall B. \text{plurality\_exists } B \equiv \exists x. \forall x'. (x \neq x') \supset |B|_{x'} < |B|_x \]

Where \( B \) is a bag\(^1\), \(|B|\) represents its cardinality, and \(|B|_x\) represents the count of \( x \) in \( B \).

---

\(^1\)Essentially a bag is a set without absorption. \([a, a, b] = [b, a, a], \) but \([a, b] \neq [a, a, b] \)
From these we define the following functions:

\[
\forall B. \text{majority } B = \epsilon x. |B| < 2|B|_x
\]

\[
\forall B. \text{plurality } B = \epsilon x. \forall x'. (x \neq x') \supset |B|_{x'} < |B|_x
\]
The property we need to prove is

\[ \forall B. \text{majority\_exists } B \supset (\text{majority } B = \text{plurality } B). \]
The first step was to show that

\[ \forall B. majority\_exists\ B \supset plurality\_exists\ B \]

For this, we needed to prove the following lemma:

\[ \forall B.\text{FINITE}\ B \supset (\forall x\ y. (x \neq y) \supset |B|_y \leq (|B| - |B|_x)) \]

From this lemma, coupled with rewriting the right conjunct of \textit{majority\_exists} to

\[ \exists x. (|B| - |B|_x) < |B|_z, \]

and then using transitivity of '<' and '<=' we can establish the existence of plurality from the existence of majority.
In order to show the equivalence between *majority* and *plurality* we needed to establish uniqueness from existence (i.e. if it exists then its unique). This allowed us to substitute in one side of the equation and then show that the chosen value satisfied the predicate embedded in the other.\(^2\)

\(^2\)Thanks to Brian Graham of the University of Calgary for submitting his methods of dealing with the HOL choice operator (‘\(e\)’ or ‘\(@\)’) to the info-hol mailing list.
Once this was done we looked at proving some other simple facts about voting which may be useful in the analysis of fault-tolerant architectures. Specifically, we proved the preservation of majority for a few common reconfiguration schemes.

- Graceful Degradation
- Perfect Spares
- Imperfect Spares

Of course, we neglected one of the more difficult aspects of reconfiguration, namely that of correctly identifying the faulty channel. All that we have done is prove a little bit of common sense.
Graceful Degradation

The simplest reconfiguration strategy is graceful degradation. This consists of removing a faulty channel and continuing processing with one less channel of redundancy. The proof for this case showed that a majority is preserved if a non-majority element is removed from consideration.

First we show existence

\[ \forall B. \forall x. \text{majority_exists } B \supset (x \in B) \supset (x \neq \text{majority } B) \supset \text{majority_exists } (B - x) \]

This essentially reduces to showing

\[ |B| < 2|B|_{x'} \supset (|B| - 1) < 2|B|_{x'}. \]

From existence we get uniqueness so we can then show

\[ \forall B. \forall x. \text{majority_exists } B \supset (x \in B) \supset (x \neq \text{majority } B) \supset (\text{majority } B = \text{majority } (B - x)) \]
Perfect Spares

Sometimes, in addition to removing a faulty channel, a good channel is added to the configuration. To capture this scenario, we showed that the insertion of the majority element to a bag preserved both existence and value of the majority.

\[ \forall B. \text{majority\_exists } B \supset \\text{majority\_exists } ((\text{majority } B) \odot B) \]

\[ \forall B. \text{majority\_exists } B \supset (\text{majority } ((\text{majority } B) \odot B) = \text{majority } B) \]
Imperfect Spares

Finally, recognizing that it is possible for spares to fail, it was shown that the removal of a non-majority (e.g., failed) element coupled with the addition of an arbitrary element (of the proper type) also preserves both existence and the value of majority.

\[ \forall B. \text{majority} \_\text{exists} \ B \supset \forall x \ x'. (x \in B) \supset \\
(x \neq \text{majority} \ B) \supset \\
\text{majority} \_\text{exists} \ (x' \ominus (B - x)) \]

\[ \forall B. \text{majority} \_\text{exists} \ B \supset \forall x \ x'. (x \in B) \supset \\
(x \neq \text{majority} \ B) \supset \\
((\text{majority} \ (x' \ominus (B - x))) = (\text{majority} \ B)) \]
Future Efforts

- Establish a base for reasoning about error manifestations in order to reason about Fault Detection and Isolation.
  
  When can you conclude that a redundant channel is faulty?

- Explore the effects that incorporating a plurality voter would have on the OS proofs.
  
  This would require adding assumptions concerning the behavior of faulty channels.

- Explore possible ways to incorporate reconfiguration strategies into the OS effort.
  
  How do you differentiate between a permanent and a transient fault?
Formally specifying the logic of an automatic guidance controller

David Guaspari
Odyssey Research Associates
Truth arises more readily from error than from confusion.

Francis Bacon

Novum Organum
The Penelope project:

• Interactive, incremental, tool for formal verification of Ada programs (Larch/Ada specifications).

  – Structure or ordinary text editor

  – Permits development of program and proof in concert, "reuse by replay"

• Covers large subset of sequential Ada.

• Mathematically based.
Problem: specify "logic" of experimental Automatic Guidance Control System for a 737

- Pilot requests kind and degrees of automatic assistance

- Requests may be honored, disallowed, "put on hold"

- Responses must be displayed
Work-in-progress: Larch/Ada specification

• Formal specification of Ada code

• Goals: precise; intelligible to designers and implementors

• Currently wrong, but clear

Related work

• Original code (CSC)

• Experiment in redesign (NASA)
knobs, switches

flight plan

sensors

logic

lights, windows

flight control
Some failures of informal description

1. Ambiguous: "Select" a switch vs. "select" a mode.

2. Incomplete: "CAS ENG may be engaged independent of all other AGCS modes except TIME PATH."

3. Contradictory:

   - FPA ... cannot be deselected directly.

   - [if] ... appropriate selection of the FPA SEL ... switch returns the mode to the off state ...
Larch/Ada specifications: “two-tiered”

- Mathematical part (Larch Shared Language): defines vocabulary

- Interface part (Larch/Ada): uses vocabulary to specify code
Example: specifying executable addition

Mathematical part: defines mathematical $+$ on $\text{Int}$, the (infinite) domain of mathematical integers

Interface part: Specifying evaluation of $x+y$

- Type integer is "based on" $\text{Int}$.

- Return value $(x + y)$ if

$$\text{min} \leq (x + y) \leq \text{max}.$$ 

No side effects.

- Otherwise, raise numeric_error. No side effects.
The mathematical part

States: AGCS_state, Sensor_state, etc.

Actions:

\{alt\_eng\_switch,\ldots, alt\_eng\_knob(i),\ldots,\nonumber\ \alt\_capture,\ldots\}\}

Modes:

\{alt\_eng,fpa\_sel,vert\_path,\ldots\}\}

Transition operation:

\[\text{AGCS\_state, Action, \ldots} \rightarrow \text{AGCS\_state}\]

Observers: active2d, display, \ldots
Building mathematical part (the AGCS states)

AgcsStructure : trait

AGCS_state record of
  (on: Bool,
   modes: Set_of_modes,
   engaged: Engagement_status,
   setting: Value_settings,
   window: Window_array)

includes Set(Mode,Set_of_modes)

... introduces
  transition:
    AGCS_state, Action, Sensor_state,
    Flight_plan \rightarrow AGCS_state
  initial_on_state: \rightarrow AGCS_state

asserts

...
Description of mode changes caused by switches:

- Is the mode directly deselectable?

- What mode changes result?

- Under what conditions is the mode directly selectable?

- What mode changes result?
Building mathematical part (mode changes)

HorPathSwitch : trait
   includes SwitchShell{hor_path}
   asserts for all
      [agcsmodes: Set_of_modes,
      pl: Flight_plan,
      sens: Sensor_state]

   hor_path_deselectable
   hor_path_selectable(agcsmodes,pl) =
      (auto ∈ agcsmodes) ∧ active2d(pl)
   hor_path_selection_result(agcsmodes,sens,pl) :
      [hor_path] ∪ [cas]
   hor_path_deselection_result(agcsmodes) =
      [tka_sel] ∪ [cas]
Intuitive description of window status (*chosen* vs. *current*):

- The $w$-knob makes the corresponding $w$-window chosen.

- Any action selecting the $w$ mode makes the $w$-window chosen.

- Any action deselecting the $w$ mode makes the $w$-window current.

- Any other action leaves the status of the $w$-window unchanged.
Building the mathematical part (window changes)

StatusShell : trait
imports AgcsStructure
introduces
    #.component :
        Window_array -> Window_status
    md: -> Mode
    knob : Value -> Action

asserts for all [agcs:AGCS_state, ...]
abbreviation
    agcs' == transition(agcs,act,sensor,plan)
    agcs'.window.component =
        if md ∈ agcs'.modes - agcs.modes
            then chosen
        elsif md ∈ agcs.mode - agcs'.modes
            then current
        elsif act = knob(i) then chosen
        else agcs.window.component

Example: StatusShell{alt,alt_eng,Airspeed}
Design of the code:

- Packages panel_logic, display_manager, sensor_data, flight_plan, flight_control.

- State of panel_logic based on AGCS_state, etc.

- Actions $\rightarrow$ procedures of panel_logic:
  - read state of panel_logic, sensor_data, flight_plan
  - modify states of panel_logic, display_manager, flight_control

- Consistent with polling, interrupts, etc.
Specifying the code:

```
--| WITH TRAIT AgcsLogic, AgcsProperties,
--|     LogicalDisplay
--| WITH sensor_data, flight_plan,
--|     display_manager, flight_control

with sensor_data_types; use sensor_data_types;

package panel_logic
   --| BASED ON AGCS_state
   --| INVARIANT
   --| panel_logic.on -> good(panel_logic)
   --| INITIALLY not panel_logic.on
   ...

end panel_logic;
```
procedure att_cws_switch;

--| WHERE

--| GLOBALS IN  panel_logic
--| GLOBALS OUT display_manager,
--|     flight_control,
--|     panel_logic

--| IN panel_logic.on

--| OUT panel_logic =
--|     transition(IN panel_logic,
--|             att_cws_switch,*,*
--|     OUT FORALL ss: Sensor_state::
--|     look(display_manager,ss) =
--|     display(panel_logic,ss)
--|     OUT FORALL md:mode ::
--|     fc_engaged(md,flight_control) =
--|     engaged(md,panel_logic)
--| END WHERE;
procedure turn_on_agcs
--| WHERE
...
--| OUT panel_logic = initial_on_state
...
--| END WHERE;
Verification of Floating-Point Software

D. N. Hoover
Odyssey Research Associates, Ithaca NY

Abstract

Floating point computation presents a number of problems for formal verification. Should one treat the actual details of floating point operations, or accept them as imprecisely defined? or should one ignore round-off error altogether, and behave as if floating point operations are perfectly accurate? There is the further problem that a numerical algorithm usually only approximately computes some mathematical function, and we often do not know just how good the approximation is, even in the absence of round-off error.

ORA has developed a theory of asymptotic correctness which allows one to verify floating point software with a minimum entanglement in these problems. We describe this theory and its implementation in the Ariel C verification system, also developed at ORA. We illustrate the theory using a simple program which finds a zero of a given function by bisection.
Verification of Floating-Point Software

Douglas Hoover

Odyssey Research Associates, Inc.
Difficulties

• Machine real arithmetic does not have nice mathematical properties

• Doesn’t match ideal arithmetic (overflow, round-off, underflow)

• Programs don’t satisfy the specification we’d like them to
Asymptotic Correctness

- Specify "ideal behavior" of the program (e.g. "program computes the square root of its input")
- Verify that if program is run on a sequence of machines converging to perfect accuracy, then program's behavior converges to ideal behavior
Advantages of the Asymptotic Approach

• Machine real arithmetic can be specified loosely

• Specifications can be written in terms of ideal behavior

• Verification does not require roundoff error analysis

• Verifies logical correctness — absence of “bugs” from inaccuracy of machine arithmetic that are not related to error magnitude.
Nonstandard analysis

\[ \mathbb{R} \subseteq \mathcal{R} \]

Standard part map

\[ st : \mathcal{R} \to \mathbb{R} \]

rounds off a finite nonstandard real to an infinitely close standard real.

Continuity

\( f \) is continuous at \((a_1, \ldots, a_n)\) if

\[ st(f(a_1, \ldots, a_n)) = f(st(a_1), \ldots, st(a_n)) \]

Differentiation by algebraic manipulation

Let \( st(\epsilon) = 0, \epsilon \neq 0 \). For all standard \( x \),

\[
\frac{d(x^2)}{dx} = st\left( \frac{(x + \epsilon)^2 - x^2}{\epsilon} \right) = st\left( \frac{2\epsilon x + \epsilon^2}{\epsilon} \right) = st(2x + \epsilon) = 2x
\]

\[ \sqrt{\ } \]
Nonstandard Analysis

- Asymptotic approach can be formalized naturally in nonstandard analysis using infinitesimals

- Primitive operations are assumed to return values which are infinitely close to the ideal values when the arguments and ideal answers are finite

- Programs are specified to have behaviors infinitely close to ideal behavior when inputs are finite
Finding Roots of a Continuous Function

- `find_zero` searches for a root of a user-supplied function $F$ by bisection.

- At each iteration, it tests to see if the values of $F$ at the left endpoint and the midpoint are of opposite sign, and changes one of the endpoints to the midpoint so as to keep a root between the two endpoints.

- The program terminates when it finds a root or when it reaches a user-supplied bound on the number of iterations.
float find_zero(left0,right0,maxit)
float left0,right0;
int maxit;
{
    float left,right,center;
    float cval,lval0,rval0;
    int numit;

    numit = 0;

    lval0 = F(left0);
    rval0 = F(right0);

    left = left0;
    right = right0;
    center = (left + right)/2.0;
    cval = F(center);

    while(cval != 0.0 && numit < maxit) {
        if (lval0 * cval < 0)
            right = center;
        else
            left = center;
        center = (left + right)/2.0;
        cval = F(center);
        lval0 = F(left);
        numit = numit + 1;
    }

    return(center);
}
Specification of find_zero

IF F is continuous and find_zero is started up with

- left0 and right0 not "large";
- maxit "large";
- $F(left0)$ and $F(right0)$ of opposite sign

THEN find_zero terminates normally (i.e. without an exception) and the value output is "close to" some zero of F.
Attempted Verification

• Proof of termination is easy.

• Proof that termination is normal is a bit harder. Must prove that no overflow happens. To prove this, must prove that the values of the endpoints stay in some range of numbers which are not “large”.

Odyssey Research Associates, Inc.
How would we prove that the program returns an approximation to a root?

- Prove when the program terminates, the endpoints are “close”. This follows from the fact that the program halves the interval a “large” number of times.

- Prove there’s always a root between the endpoints. This should follow from the way the program decides whether to move the left endpoint or the right. From this we’d get center “close to” a root.

Unfortunately, it’s not true that there’s always a root between the endpoints.
The Bug

• In the test statement, can have lval0 and cval of opposite sign, but have the product underflow to 0. This causes the program to move the wrong endpoint.

• Tests bear out this bug.
Possible Fixes

Several ways to fix this bug

- Change test to

\[(lval0 < 0 \&\& cval \geq 0) \lor (lval0 \geq 0 \&\& cval < 0)\]

- Change test so instead of always testing left endpoint against midpoint, it always tests the endpoint with the larger value of \(F\) against the midpoint. This doesn't necessarily keep a root between the endpoints, but it delivers an approximation to a root anyway.
Ariel

- Verification system for subset of C including real arithmetic and some UNIX system calls.
- Implements nonstandard formalization of the asymptotic approach.
Semantic Verification

- Ariel verifies programs by generating a description of the program's denotation in a higher-order language (the Clio metalanguage).

- Specifications are statements about the denotation in the Clio metalanguage.

- Verification is a proof of the specification directly from the description of the denotation in Clio theorem prover.

- Specifications can be any statement about the program's denotation which can be expressed in the Clio, including termination.
C Semantics

• A “run” of the program is modeled as a sequence of events

• Events are:
  – the event of going into a certain state
  – terminating and returning a value
  – terminating and returning no value
  – raising an exception
  – an “unknown” event

• The semantics of the program is expressed as a collection of axioms saying which sequences of events can happen in the course of executing the program.
Sample Verifications

- **ZBRENT** — a program which finds zeros of a continuous function by bisection

- **SWAP** — a very simple program to swap the contents of 2 locations which contains a surprising bug

- **HOSTILE BOOSTER** — a suite of programs, developed by Applied Technology Associates for SDIO, that estimate hostile booster trajectories. This verification is currently in progress.

- **SECURE DEVICE DRIVER** — specification and verification of security for an Ethernet device driver. Currently in progress.
C Formal Verification with Unix Communication and Concurrency

D. N. Hoover
Odyssey Research Associates, Ithaca NY

Abstract

This talk reports the results of a NASA SBIR project in which we developed CSP-Ariel, a verification system for C programs which use Unix system calls for concurrent programming, interprocess communication, and file input and output. This project builds on ORA's Ariel C verification system by using the system of Hoare's book Communicating Sequential Processes to model concurrency and communication. The system runs in ORA's Clio theorem proving environment. We outline how we use CSP to model Unix concurrency, and sketch the CSP semantics of a simple concurrent program. We discuss plans for further development of CSP-Ariel.
C Formal Verification with
Unix communication and concurrency
(NASA SBIR)

Aim: Verification system for

- C programs
- Unix system calls
- concurrent programming (fork, wait, exit, pipe)
- file and device i/o (read, write, open, close).
Example program.

```c
void producer();
void consumer();
int pipedes[2];

void main()
{
    int id;

    if (pipe(pipedes) == -1) return;

    id = fork();
    if (id == -1) return;
    if (id == 0) consumer();
    else producer();

    return;
}

void producer()
{
    char c;
    int status;

    while (read(0, &c, 1) != 0) /* 0 = standard input filedes */
    { 
        write(pipedes[1], &c, 1);
        close(pipedes[1]);
        exit(wait(&status));
    }

void consumer()
{
    char c;

    close(pipedes[1]); /* so that pipe read will fail when producer
    closes its write end of pipe */
    while ( read(pipedes[0], &c, 1) != 0)
    { 
        write(1, &c, 1); /* 1 = standard output filedes */
        exit(0);
    }
```
Example Program Schematic

(stdin) -> Main 

stdout

Main 

pipe

fork

producer 

pipe 

consumer
Technical Approach

- C semantics via Ariel operational semantics (pre-existing)

- Unix communication and concurrency semantics via Hoare's CSP
CSP (Communicating Sequential Processes)


- An algebraic language for describing systems of processes with synchronous communication.

- Objects of the language are processes and events.

- Processes resemble state machines, events the input alphabet. Deterministic and nondeterministic processes.

- Processes participate in events and are transformed by them.

- Synchronous communication by participation in shared events.
Unix modeling

- Unix processes, files, pipes, and certain system tables are modeled as deterministic CSP-processes.

- Forking, pipe creation, file opening and closing, I/O, waiting, and exiting are modeled as events.
Example: Asynchronous pipe communication

Sending process A, pipe P, receiving process B.

\[
\begin{align*}
A & \xrightarrow{\text{Write}(s)} A' \\
P() & \xrightarrow{\text{Write}(s)} P(s) \\
P(t) & \xrightarrow{\text{Read}(t)} P() \\
B & \xrightarrow{\text{Read}(s)} B'(s)
\end{align*}
\]
Processes transformed by events

\[
\begin{align*}
P &\quad \text{Read(s)} \quad Q(s) \\
P &\quad \text{Fork} \quad \text{Parent} || \text{Child} \\
P &\quad \text{Exit} \quad \text{RUN}
\end{align*}
\]
Verification method

- C program given

- Ariel front end generates Caliban expression for abstract syntax tree of program.

- Ariel C semantics plus Unix system call semantics define denotation of a C program and associated files inside operating system as a CSP process.

- Internal operations of systems of processes hidden by CSP concealment operation.

- We reason about the resulting CSP process in Clio. Main tools are induction on traces (event sequences) of processes, and algebraic laws of CSP. Clio is a very general theorem prover, and we are not limited in the kinds of properties we can prove about processes.
Producer as a CSP process

Read char from stdin

Read " "
(Stdin closed)

Close pipe

Read "c"

Write char to pipe "c"

Write "c"

Wait for child

Wait

Exit

Exit

Run
Hiding events:

Overall process with non-I/O events hidden.

RUN

Read "c"
CONTENTS <- C:CONTENTS

Read a char from stdin
CONTENTS

Write "head (CONTENTS)"
CONTENTS <- tail (CONTENTS)

Write a char to stdin
CONTENTS

Write(head(CONTENTS))
CONTENTS <- tail(CONTENTS)

CONTENTS = ""

RUN
CSP-Ariel Development Plan

• C semantics via Ariel symbolic interpreter (existing)

• Unix communication and concurrency semantics via deterministic CSP (initial work completed).

• Extensions to support network communication planned (sockets).

• Nondeterministic CSP and event concealment for specification and modularity (planned)

• Graphic specification support using Romulus interface (planned)
Clio, Caliban, and, Ariel

- Ariel is a semantic verification system for a subset of C, written in Caliban and the Clio metalanguage. Floating point, overflow support via asymptotic correctness.

- Caliban is a lazy, purely functional language based on recursive equations and pattern matching.

- Clio is a higher-order logic theorem prover. Caliban is its term definition language. Clio's main proof methods are induction on Caliban definitions, term rewriting, and case splitting.
This workshop was organized and chaired by Ricky W. Butler of NASA Langley Research Center.

The goals of the workshop were: (1) Define and characterize the verification problem for ultra-reliable life-critical flight control systems and the current state of the practice in industry today, (2) Determine the proper role of formal methods in addressing these problems, and (3) Assess the state of the art and recent progress toward applying formal methods to this area.

Attendees included NASA personnel, researchers from the four supporting contract organizations, RSRE personnel, invited speakers, and representatives from other government research organizations with interests in formal methods.